CV-Final-Project-Binocular Stereo

- **Group Members**

2253744 Juekai Lin	2253215 Yao Liu	2250397 Cheng Qin

- **Instructor: Lin Zhang**

1. Project Description

1.1. Project Introduction

The task of this project is to develop a binocular stereo camera and use it to get real-time depth streams. In addition, based on one stereo frame, you need to output a piece of 3D point cloud.

1.2. Project Requirements

1. For hardware, you need to buy two cameras and form a binocular stereo camera.
2. Calibrate the stereo camera.
3. Use such a camera to capture real-time depth streams.
4. Capture one stereo frame, output and visualize the corresponding piece of 3D point cloud.
5. (*Prolongation) For a 3D object, by capturing its multi-view stereo frames, reconstruct its full 3D model.

2. Project Structure

For this project, we first needed to purchase two cameras and assemble them into a binocular stereo camera system. Then the camera is calibrated, which mainly includes obtaining the internal parameters of the camera (focal length, main point position, etc.) and external parameters (relative position and attitude of the two cameras). A binocular camera is used to capture a real-time stereoscopic image stream, calculate the parallax map of each frame image, and convert it into a depth stream to form a continuous depth information stream. Then a single frame stereoscopic image is obtained, corresponding 3D point cloud data is generated by depth map and camera parameters, and the output is visualized. Finally, a three-dimensional image of a three-dimensional object is taken from multiple perspectives, and the complete three-dimensional model of the object is reconstructed by multi-perspective stereo matching and fusion technology.

These steps are closely linked and progressive: hardware preparation provides the basis for stereo vision; Camera calibration ensures the accuracy of internal and external parameters and determines the accuracy of depth calculation. Real-time deep stream capture connects hardware and applications to provide subsequent data support; Single frame point cloud generation verifies the quality of depth data. Multi-view reconstruction integrates multi-frame information to generate a complete 3D model.

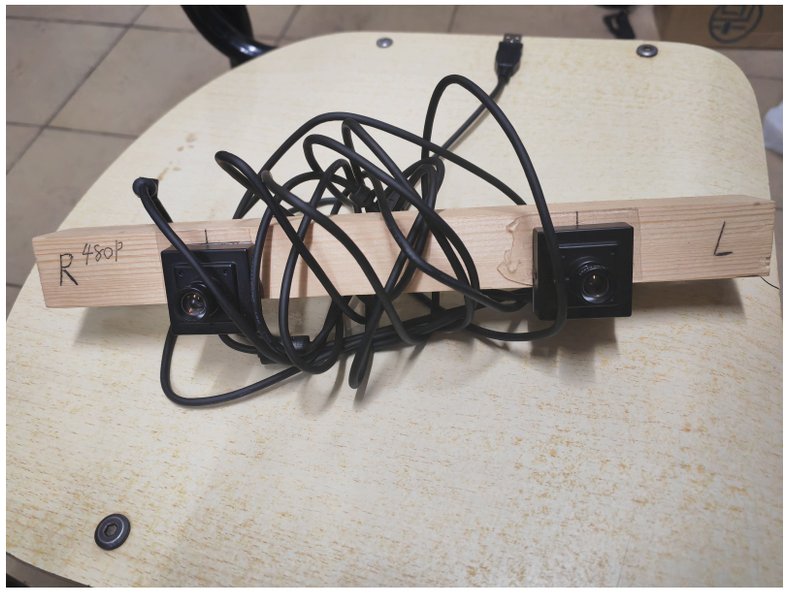
The following is the general file directory of our project, and each folder stores the relevant files of each step:

```
Binocular-Stereo
    ├── Calibration
    ├── RealTimeDepthStream
    ├── PointCloudVisualization
    ├── MultiView3DReconstruction
    ├── REPORT.pdf
    └── README.md
```

3. Hardware Device Preparation

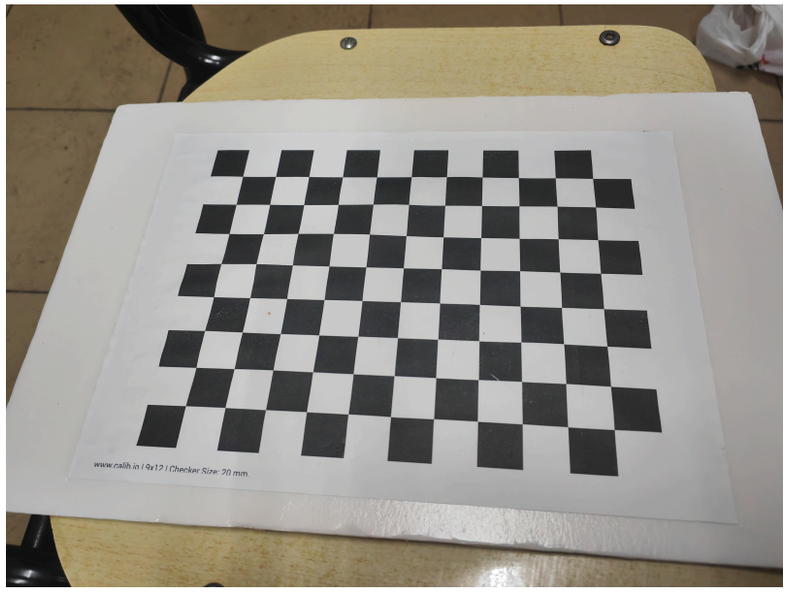
3.1. Two Cameras

In this project, we first need to purchase two cameras with USB interfaces and fix them on a rigid bracket, as shown below, to simulate human "binocular" (i.e., L position and R position). In the process of fixing, in order to make the two cameras in addition to the horizontal displacement, there is no other obvious attitude difference, the horizontal distance between the two cameras can generally be between 10-20 cm, in our project, the horizontal distance roughly selected is about 15cm.



3.2. CheckerBoard

In this project, we also needed to calibrate the internal and external parameters of the two cameras using a correction checkerboard. CheckerBoard is usually composed of black and white squares. In this project, three squares are provided under the Calibration\CheckerBoard directory. The side length of the squares can be determined according to the actual situation (such as 2cm or 3cm). We need to print the checkerboard and fix it on a flat surface to take multiple sets of checkerboard images at different angles and positions.



4. Calibrate the Stereo Camera

The purpose of binocular camera calibration is to obtain the internal and external parameters of the two cameras, which are used to correct the camera distortion and determine the relative position and attitude between the two cameras. The followings are the relevant documents for this section:

```
Calibration
    ├── CheckerBoard
    ├── stereo-imgs-480P
    |   ├── left
    |   ├── right
    |   ├── CalibrationParameterScript.m
    |   ├── Figure1.fig
    |   ├── Figure2.fig
    |   ├── Parameters.txt
    |   └── stereoParams.mat
    ├── CalibrationPicture.py
    └── FindCamera.py
```

4.1. Correlation Principle

The binocular vision system needs to calibrate internal and external parameters in order to achieve accurate image matching, parallax calculation and depth measurement. Internal parameter calibration (including focal length, main point position, distortion coefficient, etc.) is used to describe the imaging characteristics of each camera. External parameter calibration (rotation matrix R and translation vector t) describes the relative position and orientation relationship between two cameras.

- **Calibration theory of binocular external parameters**

Let us denote the left camera of the physical stereo system as Cam_l with the camera coordinate system C_l , and the right camera as Cam_r with the camera coordinate system C_r. Suppose the extrinsic parameters between C_l and C_r are a rotation matrix $R \in R^{3\times3}$and a translation vector $t \in R^3$. This means that if the coordinates of a 3D spatial point in the C_l coordinate system are p_l, its coordinates in the $C_r$coordinate system are p_r, which satisfies the equation:

$$p_r = Rp_l + t$$

Among them, R is the rotation matrix from C_l to C_r and t is the translation vector from C_l to C_r.

- **External parameter calibration method using calibration board**

The binocular camera was used to capture multiple sets of calibration plate images, and the image sets corresponding to the left and right eyes were obtained $\{I_l^i, I_r^i\}_{i=1}^m$, and the world coordinate $\{p_j\}_{j=1}^n$ of each intersection on the calibration board is known.

Based on the method in Chapter 10 (which will not be repeated here), the internal parameters of the left and right cameras are calibrated respectively, and the following results are obtained:

```
1. Internal parameter matrix $K_l$ and distortion coefficient of the left camera.
1. Internal parameter matrix $K_r$ and distortion coefficient of the right camera.
1. External parameters of the left and right cameras in each set of images relative to the
world coordinate system $(R_l^i,t_l^i)$ and $(R_r^i,t_r^i)$.
```

- **Calculation of binocular parameters**

According to the projection relationship of point p of the calibration plate, there are:

$$p_l^i = R_l^i p + t_l^i$$
$$p_r^i = R_r^i p + t_r^i$$

Simultaneous and simplified, we can get:

$$R = (R_r^i)^{-1} R_l^i,\ t = t_r^i - (R_r^i)^{-1} t_l^i$$

We can then estimate the rotation matrix R and the translation vector t from the observations of multiple sets of data.

- **Optimized calibration results**

The calibration accuracy will be affected by the measurement error, so the optimization algorithm is introduced to further adjust the parameters. The common methods are Gaussian-Newton method or Levenberg-Marquardt method, which is not overdescribed here.

- **Parameter initialization and solution**

The initial estimates of R and t can be calculated from the calibration results of a single set of images:

$$\{(R_k, t_k)\}_{k=1}^m$$

Using the initial values of multiple sets of data, the final R, t is solved by nonlinear least square optimization

4.2. Practice Process

- **Get the image data set**

In binocular calibration, both cameras need to shoot at the calibration board at the same time, and the image pair needs to be saved in sequence, so we first need to write a script that allows both cameras to take pictures at the same time, while storing in sequence in the folder of the external target:

```
# CalibrationPicture.py:  Allows both cameras to take pictures at the same time.
import cv2
import os
print("OpenCV version:", cv2.__version__)

# Get the current directory of the script
current_file_path = os.path.abspath(__file__)
current_dir = os.path.dirname(current_file_path)
print("Current script directory:", current_dir)

# Create folders (if not exist)
def create_folders(left_folder, right_folder):
    os.makedirs(left_folder, exist_ok=True)
    os.makedirs(right_folder, exist_ok=True)

# Get the next image index
def get_next_image_index(left_folder, right_folder):
    # Get all files from both folders
    left_files = os.listdir(left_folder)
    right_files = os.listdir(right_folder)

    # Extract image file numbers from both folders
    left_indices = [int(file.split('.')[0]) for file in left_files if file.endswith('.jpg')
and file.split('.')[0].isdigit()]
    right_indices = [int(file.split('.')[0]) for file in right_files if
file.endswith('.jpg') and file.split('.')[0].isdigit()]

    # Get the maximum number
    left_max = max(left_indices) if left_indices else 0
    right_max = max(right_indices) if right_indices else 0
```

```
    # Raise an error if the maximum numbers in both folders are different
    if left_max != right_max:
        return 0

    # Return the next index
    return left_max + 1


# Initialize cameras
def initialize_cameras(left_camera_id, right_camera_id):
    left_cap = cv2.VideoCapture(left_camera_id)
    right_cap = cv2.VideoCapture(right_camera_id)

    #left_cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1920)
    #left_cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 1080)
    #right_cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1920)
    #right_cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 1080)

    if not left_cap.isOpened() or not right_cap.isOpened():
        print("Failed to open cameras. Please check the connection.")
        left_cap.release()
        right_cap.release()
        return None, None

    return left_cap, right_cap

# Capture and save images
def capture_and_save_images(left_cap, right_cap, left_folder, right_folder, next_index):
    ret_left, frame_left = left_cap.read()
    ret_right, frame_right = right_cap.read()

    if ret_left and ret_right:
        # Save the images
        left_path = os.path.join(left_folder, f"{next_index}.jpg")
        right_path = os.path.join(right_folder, f"{next_index}.jpg")
        cv2.imwrite(left_path, frame_left)
        cv2.imwrite(right_path, frame_right)

        print(f"Saved successfully: Left Image {left_path}, Right Image {right_path}")
    else:
        print("Failed to capture images. Please check the cameras.")

# Main function
def capture_stereo_images():
    # Set paths
    left_folder = os.path.join(current_dir, "stereo-imgs-480P/left")
    right_folder = os.path.join(current_dir, "stereo-imgs-480P/right")

    # Create folders
    create_folders(left_folder, right_folder)

    # Initialize cameras
    # If there are more than two network heads, change id to try
    left_camera_id = 1  # Left camera ID
```

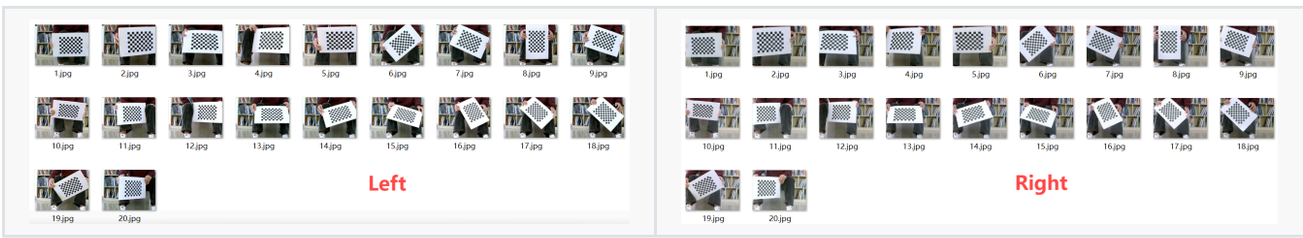
```
    right_camera_id = 2  # Right camera ID
    left_cap, right_cap = initialize_cameras(left_camera_id, right_camera_id)
    if not left_cap or not right_cap:
        print("Error: Camera not connected.")
        return

    # Get the next image index
    next_index = get_next_image_index(left_folder, right_folder)
    if next_index == 0:
        print("Error: The image indices in left and right folders are not consistent.")
        return

    # Capture and save images
    capture_and_save_images(left_cap, right_cap, left_folder, right_folder, next_index)

    # Release resources
    left_cap.release()
    right_cap.release()
    cv2.destroyAllWindows()

# Call the main function
if __name__ == "__main__":
    capture_stereo_images()
```

In the process of the experiment, we found that because the personal laptop has a camera, we need to find the corresponding camera according to the number when taking pictures. (Here, note that the number of each USB camera on each laptop is different from that of the camera on the computer.) So each time we need to run the following script to find the numbers of the two cameras we are using:

```
# FindCamera.py: Find each number of the two cameras.
import cv2

def list_cameras():
    """
    List all available camera IDs.
    """
    cameras = []
    for camera_id in range(10):  # Test for camera IDs from 0 to 9
        cap = cv2.VideoCapture(camera_id)
        if cap.isOpened():
            cameras.append(camera_id)
            cap.release()
    return cameras

def display_camera(camera_id):
    """
    Display the live feed from the specified camera ID.
    """
    cap = cv2.VideoCapture(camera_id)
    if cap.isOpened():
        print(f"Displaying camera {camera_id}. Press 'q' to close.")
        while True:
            ret, frame = cap.read()
            if not ret:
                break
```

```
            cv2.imshow(f"Camera {camera_id}", frame)
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
        cap.release()
        cv2.destroyAllWindows()
    else:
        print(f"Camera {camera_id} could not be opened.")

def main():
    """
    Main function to list cameras and display each one.
    """
    cameras = list_cameras()
    if cameras:
        print(f"Found {len(cameras)} camera(s):")
        for camera_id in cameras:
            print(f"Camera ID: {camera_id}")
            display_camera(camera_id)  # Display the camera feed
    else:
        print("No cameras found!")

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

After finding each camera, modify left_camera_id and right_camera_id in the original photo script to run the photo program, and store the two groups of pictures in the left folder and the right folder respectively:



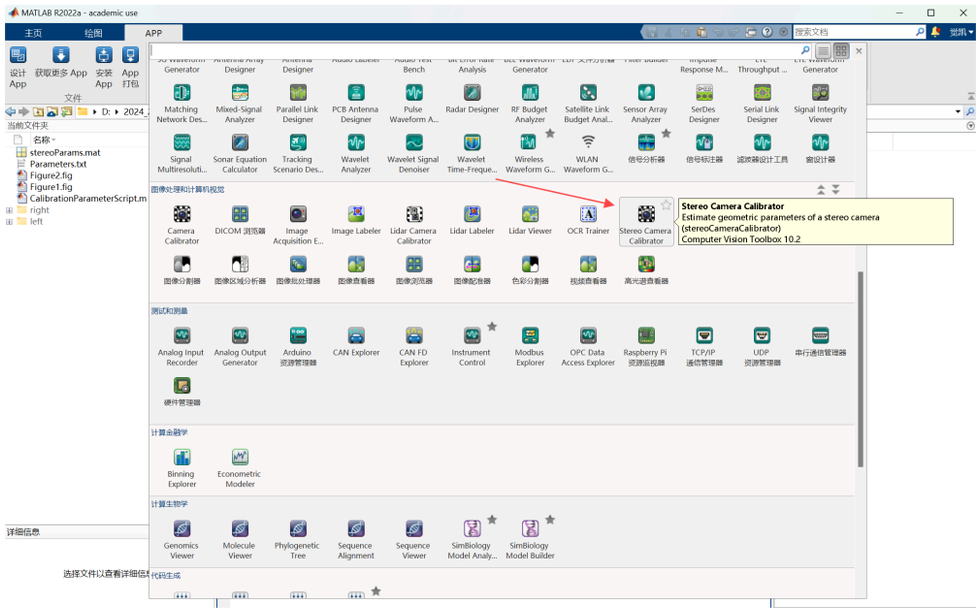
The quality of photos is **very important** for the accuracy of our final calibration results. In the process of calibration, our team conducted no less than five calibration, and summarized the following points for attention:

1. For the calibration board, be sure to ensure that there is no stain on the surface of the period, no reflection, etc., which is very important, and affects the results of the calibration. At the same time, the calibration paper we print out must be attached to a relatively hard plane, such as wood, kt board, etc., because if the relatively soft plane, the calibration paper is easy to bend and affect the final result.
2. Secondly, for the photos taken, it is necessary to ensure that all the points in the calibration plate of the left and right pictures are in the camera picture to prevent the internal points from being detected at the end. That is, when you move the camera position or move the calibration board, be sure to check whether the two pictures are taken.
3. Take as many photos of different positions and angles as possible, at least 10-20 sets of images for calibration. Of course, the quality of the picture should also be guaranteed, not fuzzy, and must pay attention to the influence of the light, the light can not be too bright or too dark, while the light intensity of the left and right cameras should be reached, it is best not to have a larger light on one side and a darker light on the other side.

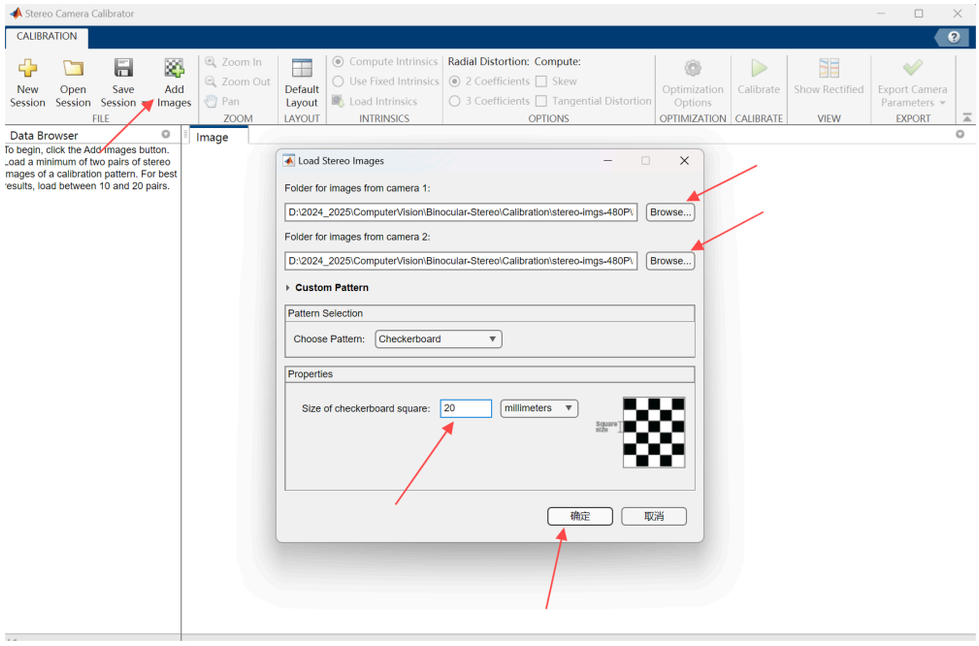
- **Assist the calibration by matlab**

Matlab software has an auxiliary binocular parameter calibration method, the specific operation is as follows:

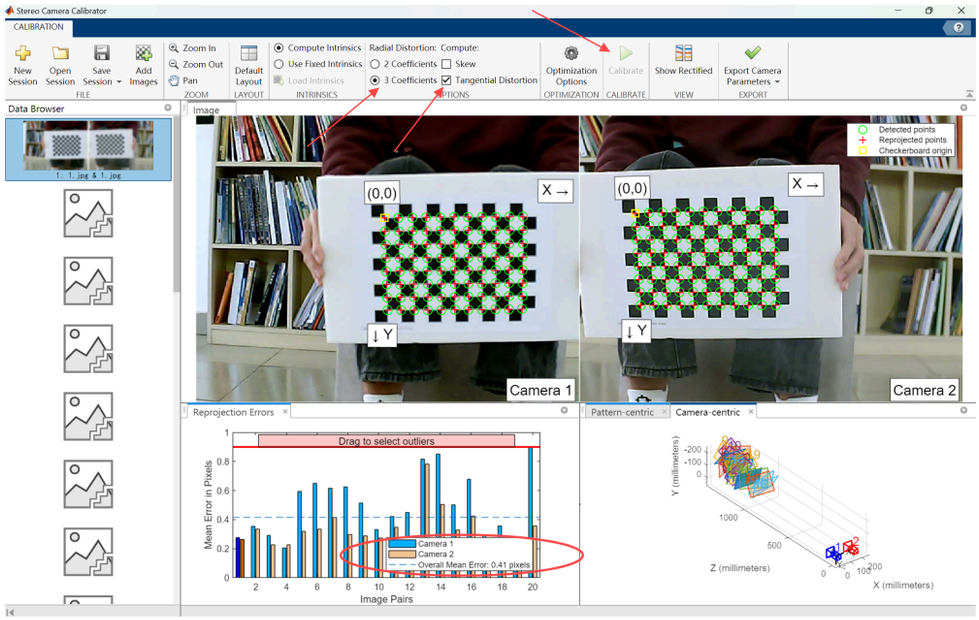
Enter the matlab software and select stereo camera calibration from the drop-down menu of the app to prepare for using the built-in binocular calibration function of matlab.



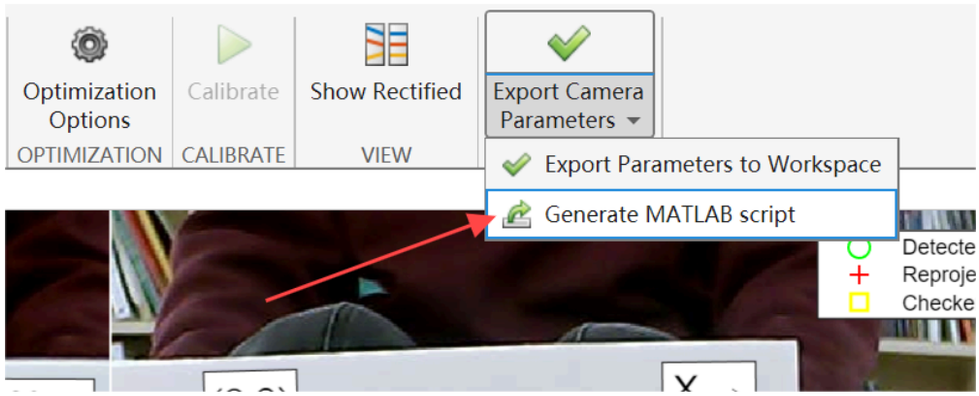
After entering this app, click "add images", select all the pictures in the left and right folders that have been taken above, and select the size parameter of the checkerboard, then we can import our previous image data for further calibration.



In the import image dataset, get ready to Calibrate, because even though the camera manual says that it is distortionless, we need to select "3 Coefficients" and "Tangential Distortion" and click the Calibrate button to calibrate it. The calibration mean error in pixels obtained at last represents the average value of calibration error. In pixel, it represents the average Euclidean distance between the position of the Angle point of the calibration board and the position of the reprojection point detected on all images, in pixel. Also shown is a Camera-centric diagram, which refers to a schematic of a camera-centered coordinate system, often used to describe the geometric relationship of cameras in computer vision. This diagram is used to show the spatial relationship between the camera and the multiple calibration plate.



After the calibration is completed, click "Export Camera Parameters" on the right and select "Generate MATLAB Script" to generate the code related to the calibration process and save it.



OK, now we have completed the whole process of calibrating the camera parameters.

Although the whole process seems clear, there are many caveats, such as the one about mean error in pixels:

Mean error in pixels is the average Euclidean distance between the position of the calibration board corner point and the position of the reprojection point detected on all images, in pixels. For the influence of subsequent calculation accuracy, mean error should be controlled within a certain range. If the value is greater than 1 pixel, it indicates that there may be a large calibration error, which may affect the subsequent binocular matching or 3D reconstruction accuracy, and it is recommended to re-calibrate. Of course, if this value can be as small as possible, the better.

4.3. Calibration Result

Our calibration code files before opening CalibrationParameterScript.m. In displayErrors(estimationErrors, stereoParams); Add a line of code below:

```
save('stereoParams.mat','stereoParams');
```

Then run this program, you can get the binocular camera calibration parameters of the file: stereoParams.mat, this file later in the construction of real-time depth stream and 3D point cloud is very useful.

```
% CalibrationParameterScript.m
% Auto-generated by stereoCalibrator app on 11-Dec-2024
```

```
%-------------------------------------------------------

% Define images to process
imageFileNames1 = {'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\1.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\10.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\11.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\12.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\13.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\14.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\15.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\16.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\17.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\18.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\19.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\2.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\20.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\3.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\4.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\5.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\6.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\7.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\8.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\left\9.jpg',...
    };
imageFileNames2 = {'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\1.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\10.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\11.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\12.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\13.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\14.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\15.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\16.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\17.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\18.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\19.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\2.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\20.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\3.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\4.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\5.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\6.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\7.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\8.jpg',...
    'E:\Binocular-Stereo\Calibration\stereo-imgs-480P\right\9.jpg',...
    };

% Detect calibration pattern in images
detector = vision.calibration.stereo.CheckerboardDetector();
[imagePoints, imagesUsed] = detectPatternPoints(detector, imageFileNames1, imageFileNames2);

% Generate world coordinates for the planar patten keypoints
squareSize = 20;  % in units of 'millimeters'
worldPoints = generateWorldPoints(detector, 'SquareSize', squareSize);

% Read one of the images from the first stereo pair
```

```
I1 = imread(imageFileNames1{1});
[mrows, ncols, ~] = size(I1);

% Calibrate the camera
[stereoParams, pairsUsed, estimationErrors] = estimateCameraParameters(imagePoints,
worldPoints, ...
    'EstimateSkew', false, 'EstimateTangentialDistortion', true, ...
    'NumRadialDistortionCoefficients', 3, 'WorldUnits', 'millimeters', ...
    'InitialIntrinsicMatrix', [], 'InitialRadialDistortion', [], ...
    'ImageSize', [mrows, ncols]);

% View reprojection errors
h1=figure; showReprojectionErrors(stereoParams);

% Visualize pattern locations
h2=figure; showExtrinsics(stereoParams, 'CameraCentric');

% Display parameter estimation errors
displayErrors(estimationErrors, stereoParams);

save('stereoParams.mat','stereoParams');

% You can use the calibration data to rectify stereo images.
I2 = imread(imageFileNames2{1});
[J1, J2, reprojectionMatrix] = rectifyStereoImages(I1, I2, stereoParams);

% See additional examples of how to use the calibration data.  At the prompt type:
% showdemo('StereoCalibrationAndSceneReconstructionExample')
% showdemo('DepthEstimationFromStereoVideoExample')
```

Running this program at the same time, the console will also output some debugging information about the binocular camera Parameters, we will integrate it out and save it as the parameters.txt file.

```
Parameters.txt
            Standard Errors of Estimated Stereo Camera Parameters
            -----------------------------------------------------

Camera 1 Intrinsics
-------------------
Focal length (pixels):   [ 1346.3086 +/- 4.7824     1349.4209 +/- 4.8069  ]
Principal point (pixels):[  285.0131 +/- 12.2600     307.7359 +/- 8.5158  ]
Radial distortion:       [   -0.6752 +/- 0.0562        1.2786 +/- 1.2235       -3.6783 +/-
10.1446 ]
Tangential distortion:   [   -0.0053 +/- 0.0017        0.0191 +/- 0.0034  ]

Camera 1 Extrinsics
-------------------
Rotation vectors:
          [    0.3072 +/- 0.0068        0.1005 +/- 0.0096       -0.0231 +/- 0.0012  ]
          [    0.6822 +/- 0.0064        0.0194 +/- 0.0089        0.1066 +/- 0.0027  ]
          [    0.0321 +/- 0.0064       -0.3670 +/- 0.0091       -0.0589 +/- 0.0015  ]
          [   -0.0381 +/- 0.0065        0.4709 +/- 0.0091        0.0652 +/- 0.0013  ]
          [   -0.6395 +/- 0.0063        0.0422 +/- 0.0087        0.0119 +/- 0.0033  ]
          [   -0.5545 +/- 0.0065        0.1021 +/- 0.0088       -0.2737 +/- 0.0029  ]
          [   -0.6630 +/- 0.0064        0.0926 +/- 0.0087        0.3577 +/- 0.0034  ]
```

```
          [   -0.3088 +/- 0.0099       -0.4130 +/- 0.0086       -1.9831 +/- 0.0029  ]
          [   -0.3237 +/- 0.0071        0.1929 +/- 0.0091        0.8107 +/- 0.0020  ]
          [   -0.2209 +/- 0.0071        0.0915 +/- 0.0093        0.7692 +/- 0.0015  ]
          [    0.5227 +/- 0.0068        0.2087 +/- 0.0090       -0.5265 +/- 0.0022  ]
          [    0.2137 +/- 0.0066       -0.1933 +/- 0.0093       -0.0645 +/- 0.0012  ]
          [    0.1156 +/- 0.0061       -0.5852 +/- 0.0091       -0.0064 +/- 0.0021  ]
          [   -0.5453 +/- 0.0063        0.0487 +/- 0.0088        0.0522 +/- 0.0029  ]
          [   -0.2141 +/- 0.0064        0.4498 +/- 0.0091        0.0861 +/- 0.0018  ]
          [    0.5844 +/- 0.0064        0.0604 +/- 0.0090       -0.0400 +/- 0.0023  ]
          [    0.1603 +/- 0.0084        0.0595 +/- 0.0098       -0.6864 +/- 0.0006  ]
          [    0.0461 +/- 0.0091        0.0296 +/- 0.0102        0.4408 +/- 0.0005  ]
          [    0.1057 +/- 0.0107       -0.0374 +/- 0.0110        1.5769 +/- 0.0006  ]
          [   -0.0463 +/- 0.0146        0.1209 +/- 0.0133       -2.8095 +/- 0.0007  ]

Translation vectors (millimeters):
          [   26.4036 +/- 10.5334    -116.9542 +/- 7.2603     1154.0228 +/- 4.0370  ]
          [   28.7177 +/- 12.0387    -208.4850 +/- 8.2810     1314.6416 +/- 4.7571  ]
          [  -12.0315 +/- 12.0127    -175.8499 +/- 8.2635     1313.5439 +/- 4.6993  ]
          [   93.3517 +/- 12.9268    -205.5105 +/- 8.9063     1412.9955 +/- 4.8353  ]
          [   42.5523 +/- 11.8498    -154.3655 +/- 8.1638     1297.1057 +/- 4.3559  ]
          [   42.3184 +/- 11.9386    -136.3738 +/- 8.2273     1307.4741 +/- 4.3846  ]
          [   71.1370 +/- 12.6539    -176.7309 +/- 8.7245     1384.9175 +/- 4.6328  ]
          [  135.1776 +/- 12.0095     -32.3454 +/- 8.3424     1320.1730 +/- 4.7364  ]
          [  104.0315 +/- 13.0155    -253.8547 +/- 8.9601     1420.1864 +/- 4.9628  ]
          [  109.5874 +/- 12.4377    -235.4490 +/- 8.5695     1357.9835 +/- 4.8112  ]
          [   25.7430 +/- 12.0337    -110.8315 +/- 8.2969     1318.9716 +/- 4.4931  ]
          [   36.2906 +/- 8.8457     -138.1727 +/- 6.0894      966.8897 +/- 3.5702  ]
          [   12.1807 +/- 11.3073    -146.9757 +/- 7.7935     1237.7840 +/- 4.4848  ]
          [   15.4319 +/- 9.5836     -146.4231 +/- 6.5928     1047.8837 +/- 3.5972  ]
          [   30.1124 +/- 10.3749    -168.7466 +/- 7.1380     1133.7496 +/- 3.8420  ]
          [   18.7247 +/- 8.7173     -138.7742 +/- 5.9968      952.6392 +/- 3.5150  ]
          [  -18.9063 +/- 11.3161     -76.5424 +/- 7.8064     1241.2205 +/- 4.2778  ]
          [   43.1876 +/- 11.3194    -202.9019 +/- 7.7905     1236.5671 +/- 4.4377  ]
          [  174.2228 +/- 11.2415    -197.3805 +/- 7.7738     1227.7043 +/- 4.6305  ]
          [  181.8103 +/- 11.3237     -13.4833 +/- 7.8815     1244.7008 +/- 4.5570  ]

Camera 2 Intrinsics
-------------------
Focal length (pixels):   [ 1358.2210 +/- 4.6506     1356.8426 +/- 4.8838  ]
Principal point (pixels):[  306.9473 +/- 10.4124     329.0460 +/- 9.0692  ]
Radial distortion:       [   -0.5808 +/- 0.0660       -5.0115 +/- 1.8821       52.1822 +/-
20.8208 ]
Tangential distortion:   [   -0.0233 +/- 0.0033       -0.0086 +/- 0.0022  ]

Position And Orientation of Camera 2 Relative to Camera 1
---------------------------------------------------------
Rotation of camera 2:[0.0192 +/- 0.0085       -0.0537 +/- 0.0110        0.0020 +/- 0.0006  ]
Translation of camera 2 (millimeters):[ -148.8616 +/- 0.1712    -2.6196 +/- 0.2164
-2.7848 +/- 1.9705  ]
```

In this way, to sum up, the first step of the project for binocular camera internal parameter calibration has been successfully completed, and it should be noted that the stereoParams.mat file has been obtained.

5. The Real-time Depth Streams

Binocular vision uses two cameras to obtain images of the same scene from different angles. By analyzing the parallax of the same scenic spot in the two images, combined with the internal and external parameters of the camera, the depth of the scenic spot is calculated.The project structure for this section is as follows:

```
RealTimeDepthStream
    ├── stereoParams.mat
    ├── RealTimeDepthStream.m
    └── RealTimeVideo.mp4
```

5.1. Correlation Principle

- **Corrected binocular system**

In our project, we will first use the corrected binocular system. In the corrected binocular system, the geometric relationship and imaging relationship of the two cameras meet the following conditions:

1. The optical axes of the two cameras are parallel and the focal length f_{rect} is the same.
2. The pixel sizes of the imaging planes of the two cameras are equal and are denoted as δ.
3. The baseline length b of both cameras is a known quantity.
4. After correction, the left and right images are aligned line by line, that is, the pixel coordinate $v_l = v_r$ in the vertical direction is satisfied.

- **The definition of parallax**

For a certain point $P_w(x, y, z)$ in space, its pixel coordinates projected on the corrected left and right eye cameras are:

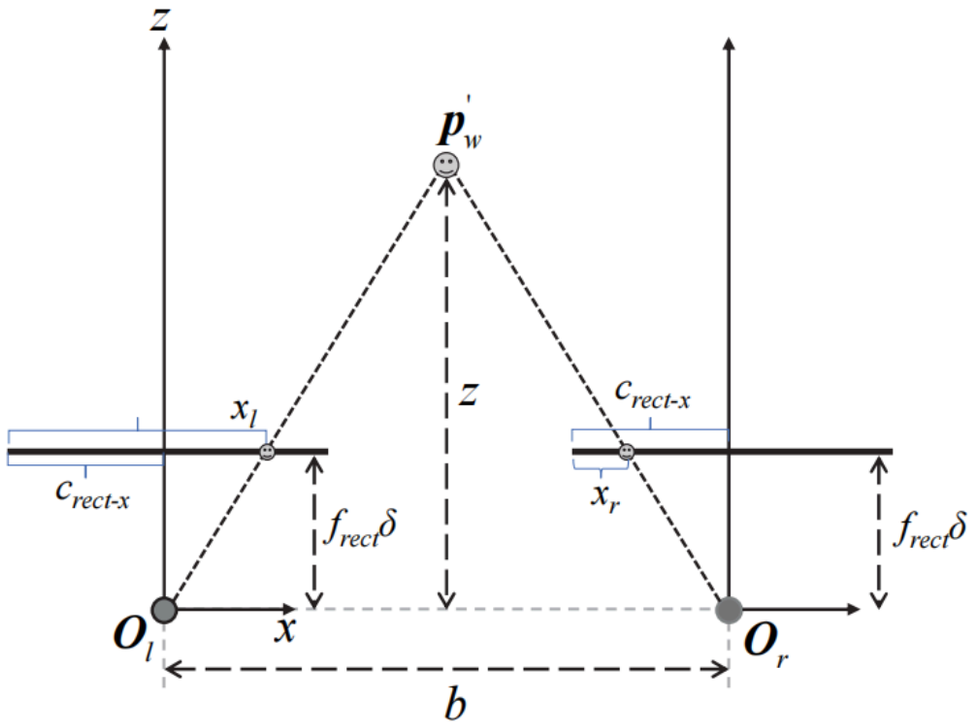
1. Left eye projection point: (x_l, y_l).
2. Right eye projection point: (x_r, y_r).

According to the nature of the correction, the vertical pixel coordinates satisfy $y_l = y_r$, so we mainly focus on the difference of horizontal pixels:

$$disparity = d = x_l - x_r$$

- **Depth calculation formula derivation**

In the corrected binocular system, the depth z is calculated based on the similar triangle principle.



According to the geometric relationship in the figure, the relationship between the projection point x_l of the left camera and the projection point x_r of the right camera and the space point is as follows:

$$x_l = \frac{f_{rect}}{z} \cdot (X + \frac{b}{2})$$
$$x_r = \frac{f_{rect}}{z} \cdot (X - \frac{b}{2})$$

Then we can calculate the parallax d to be:

$$z = \frac{f_{rect} \cdot b}{d}$$

In this formula, the parameters are:

1. z: The depth of the site to the camera (usually in meters).
2. f_{rect}: Corrects the focal length of the camera (pixel), that is, the focal length parameter in the internal parameter matrix.
3. b: Baseline length between cameras (in millimeters or meters).
4. d: Parallax (pixel).

With these basic formulas in hand, we understand the fundamentals and construction process of a basic depth map, and then we can actually write code to generate a real-time depth flow map.

5.2. Practice Process

- **Preparation Packages**

Before we start writing the code, we first need to download the following dependency packages in the Additional Features Explorer in matlab: *Computer Vision Toolbox, Image Acquisition Toolbox, Image Processing Toolbox*.

If you need other dependency packages in the process of writing and running code, download them according to the relevant requirements.

- **Coding Realization**

First of all, we need to load the binocular camera parameters in the stereoParams.mat file, that is, the data obtained in the camera calibration before, which contains the key calibration parameters of the binocular camera, such as internal parameters, external parameters, distortion parameters, baseline length, etc.

```
% import stereoParams
stereoParams = load('stereoParams.mat');
stereoParams = stereoParams.stereoParams;
```

Since we want to do real-time image acquisition and processing, we need to initialize the left and right cameras (note that we should also process according to the camera number on our computer) and set the resolution to 640x480. At the same time, set the camera to capture frames continuously, set the image to return to RGB format, and complete the real-time image acquisition preparation.

```
% Initialize camera
leftCam = videoinput('winvideo', 1, 'YUY2_640x480');
rightCam = videoinput('winvideo', 2, 'YUY2_640x480');
set(leftCam, 'FramesPerTrigger', Inf);
set(rightCam, 'FramesPerTrigger', Inf);
set(leftCam, 'ReturnedColorspace', 'rgb');
set(rightCam, 'ReturnedColorspace', 'rgb');
start(leftCam);
start(rightCam);
```

The next step is to complete the real-time calculation of the depth of the camera. According to the steps described above, our first step should be stereoscopic correction of the image taken by the binocular camera.

```
% Stereoscopic correction of pictures taken by binocular cameras
[frameLeftRect, frameRightRect] = rectifyStereoImages(frameLeft, frameRight, stereoParams);
```

After that, we need to convert the corrected image to grayscale, which is intended to reduce the computational complexity.

```
% Convert to grayscale
frameLeftGray = im2gray(frameLeftRect);
frameRightGray = im2gray(frameRightRect);
```

Now it's time to calculate the parallax map. We use disparitySGM to calculate the parallax map and set the parallax range [0, 64] with a uniqueness threshold of 15 to improve accuracy. In this way, we can display the left and right camera images and the calculated parallax map in real time:

```
% Calculate disparity map
disparityMap = disparitySGM(frameLeftGray, frameRightGray, ...
                            'DisparityRange', [0, 64], ...
                            'UniquenessThreshold', 15);
```

Finally, we can obtain the parallax value based on the mouse coordinates (x, y). Depth is calculated using baseline length, focal length, and parallax values to display the depth of the object where the mouse is pointing in real time.

```
% Shows the depth of the object where the mouse is pointing
disparity = globalDisparityMap(y, x);
baseline = norm(globalStereoParams.TranslationOfCamera2);
focalLength = mean([globalStereoParams.CameraParameters1.FocalLength]);
depth = (baseline * focalLength) / (disparity * 1000);
set(globalDepthText, 'String', sprintf('Depth: %.2f m', depth));
```

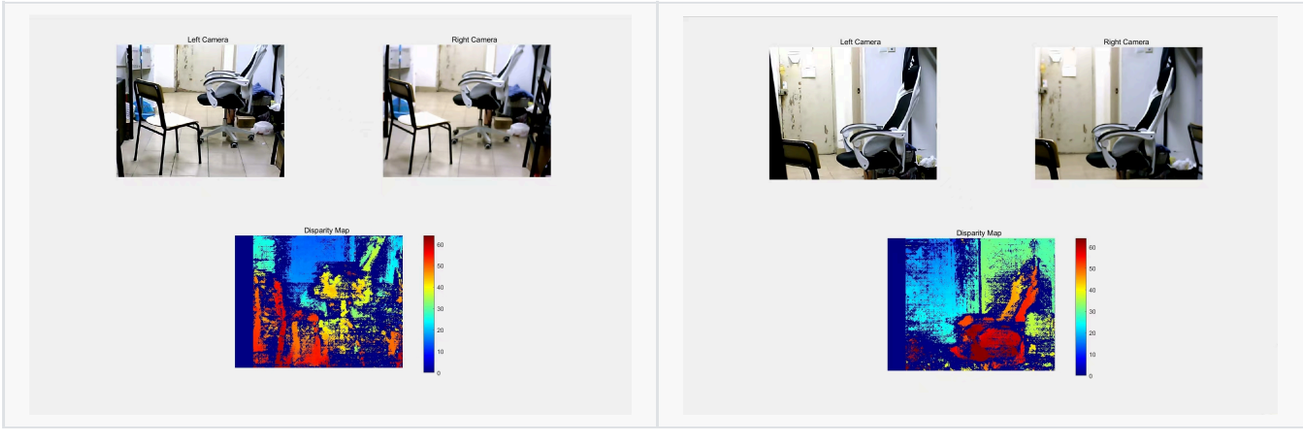
Finally, we want to set the real-time depth effect, that is, when we place the mouse over the relevant position of the video, it will show the approximate depth of that position. We use the following function to do this:

```
% mouse move callback function
function mouseMove(src, ~)
    global globalDisparityMap globalStereoParams globalAxesHandle globalDepthText;

    try
        % get current image axis
        current_axes = get(src, 'CurrentAxes');

        % display depth only on disparity map
        if current_axes == globalAxesHandle
            % get current coordinates
            pos = get(current_axes, 'CurrentPoint');
            x = round(pos(1,1));
            y = round(pos(1,2));

            % check coordinates are valid
```

```
            if ~isempty(globalDisparityMap) && ...
               x >= 1 && x <= size(globalDisparityMap, 2) && ...
               y >= 1 && y <= size(globalDisparityMap, 1)

                % get disparity value
                disparity = globalDisparityMap(y, x);

                % calculate depth
                if disparity > 0
                    % get baseline and focal length
                    baseline = norm(globalStereoParams.TranslationOfCamera2); % baseline
length mm

                    focalLength = mean([globalStereoParams.CameraParameters1.FocalLength]);
% focal length

                    % calculate depth m
                    depth = (baseline * focalLength) / (disparity * 1000);

                    % limit depth range and display
                    depth = min(max(depth, 0.1), 10);

                    % update depth text
                    set(globalDepthText, 'String', sprintf('Depth: %.2f m', depth));
                    set(globalDepthText, 'Position', [x+10, y, 0]);
                    set(globalDepthText, 'Visible', 'on');
                else
                    set(globalDepthText, 'String', 'Invalid depth');
                    set(globalDepthText, 'Position', [x+10, y, 0]);
                    set(globalDepthText, 'Visible', 'on');
                end
            else
                set(globalDepthText, 'Visible', 'off');
            end
        else
            set(globalDepthText, 'Visible', 'off');
        end
    catch e
        disp(['Mouse callback error: ' e.message]);
    end
end
```

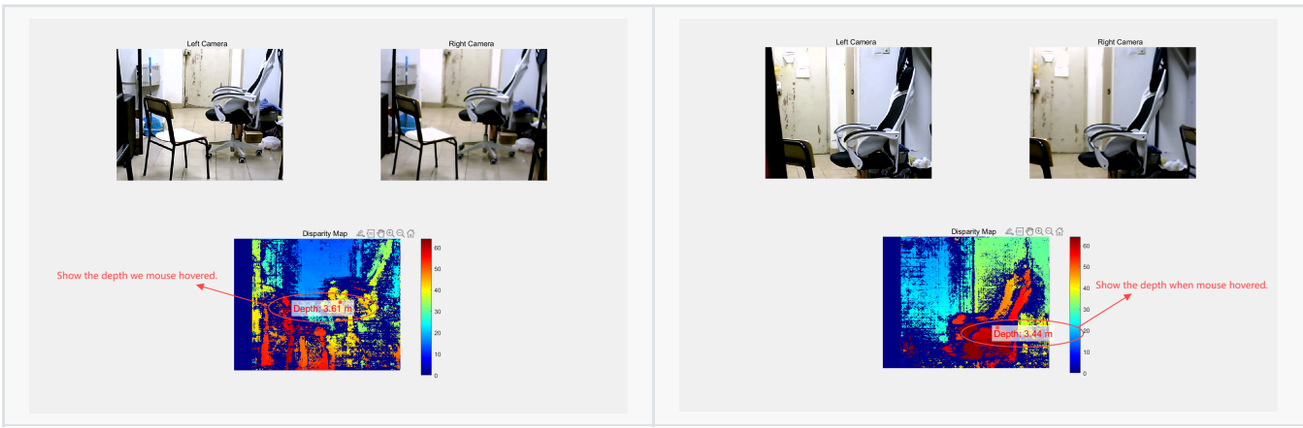
5.3. Experiment Result

After ensuring that both cameras are connected to your computer and checking the corresponding numbers for both cameras, run RealTimeDepthStream.m to create a real-time depth stream from the images captured by both cameras. Here are two examples.



From the real-time parallax map constructed above, we can find that compared with the color displayed in the parallax map and the real picture, it is found that basically the physical contour can match and calculate the effect is quite good.

Next, we can hover the mouse over the location of the live video we want to view, and the approximate depth of the location will be suspended. The depth shown is compared with the actual position depth in reality, which is also reasonable and ideal. In this way, our real-time depth flow map is successfully completed.



Here are some caveats: Parallax is calculated from the stereo matching algorithm, which represents the difference in pixel position of the same object point in two cameras.

If the parallax value is 0 or negative, it indicates that the point is not correctly matched, which may be caused by occlusion, camera itself, etc., and will be displayed as invalid point.

At the same time, in the process of judging the depth, when the calculated depth is greater than 10 meters, the depth will be cut to 10 meters, which is to avoid the depth value is too large or unreasonable, because the accuracy of the binocular camera we use cannot guarantee the depth accuracy beyond 10 meters, which can prevent the infinite increase of the depth value and the depth value is too large and has no practical significance.

We recorded a result display video RealTimeVideo.mp4 placed in the relevant folder directory, you can view the video real-time depth stream specific effects.

6. A 3D Point Cloud of a Stereo Frame

The core of generating 3D point clouds is to correct the parallax value of each pixel in the parallax map, combine the camera parameters (focal length, baseline, main point, etc.), calculate the corresponding three-dimensional space coordinates using geometric relations, and map color information to generate three-dimensional point clouds. The project structure for this section is as follows:

```
PointCloudVisualization
    ├── stereoParams.mat
    ├── test-left
    ├── test-right
    ├── PointCloudVisualization.m
    └── result.ply
```

6.1. Correlation Principle

- **From image coordinates to camera normalized plane coordinates**

The image coordinate system is a two-dimensional coordinate system in pixels, and the coordinate of a point $u = (x, y)$ represents the position of a pixel on the image. The normalized plane of a camera is a coordinate system with the optical center of the camera as the origin and in units of physical length (such as millimeters or meters), assuming that the plane is parallel to the camera imaging plane and that the point x_n on it is $z = 1$.

The conversion from image coordinate u to normalized plane coordinate x_n is completed by the following formula:

$$x_n = \frac{x - c_x}{f}, y_n = \frac{y - c_y}{f}$$

1. c_x, c_y: The principal point coordinate is the position of the optical center in the image coordinate system.
2. f: Focal length indicates the distance from the optical center of the camera to the image plane.

Finally, points on the normalized plane can be expressed as:

$$x_n = \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix}$$

- **Calculation of three-dimensional point cloud coordinates**

The three-dimensional point coordinate $p = [X, Y, Z]$ is accomplished by combining the pixel point $u = (x, y)$ on the image plane with the parallax value d, and converting the camera calibration parameters to the corrected left-eye camera coordinate system C_{rect-l}. The specific process is as follows:

1. In camera calibration, the formula is used:

$$p = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} (x - c_x) \cdot \frac{b}{d} \\ (y - c_y) \cdot \frac{b}{d} \\ f \cdot \frac{b}{d} \end{bmatrix}$$

2. To make it easier to handle multiple pixels, the above formula can be converted into a matrix form:

$$p = Q \cdot \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix}$$

Where Q is the reprojection matrix used to map pixel coordinates and parallax values directly to 3D camera coordinates:

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & \frac{1}{b} & 0 \end{bmatrix}$$

3. For each effective pixel in the parallax plot (x, y, d), its three-dimensional coordinates (X, Y, Z) are computed using matrix operations. The homogeneous coordinate results are normalized:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_h/W \\ Y_h/W \\ Z_h/W \end{bmatrix}$$

Where w is the fourth component of the homogeneous coordinate, ensuring that the result is a non-homogeneous three-dimensional coordinate.

- **Color point cloud generation and visualization**

In the second task above, we have calculated the parallax map, and next we need to calculate the three-dimensional point cloud coordinates and generate the colored point cloud.

For each effective pixel point (x, y, d), the three-dimensional coordinates are calculated using the reprojection matrix Q:

$$p = Q \cdot \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix}$$

The computed three-dimensional points (X, Y, Z) are in the left-eye camera coordinate system.

Using the pixel color value (R, G, B) of the left-eye image I_{rect-l}, map it to the corresponding three-dimensional point (X, Y, Z).

Each generated point contains three-dimensional coordinates and color information (X, Y, Z, R, G, B).

6.2. Practice Process

First of all, we need to load the binocular camera parameters in the stereoParams.mat file, that is, the data obtained in the camera calibration before, which contains the key calibration parameters of the binocular camera, such as internal parameters, external parameters, distortion parameters, baseline length, etc.

```
% read stereoParams
stereoParams = load('stereoParams.mat');
stereoParams = stereoParams.stereoParams;

% visualize extrinsics
showExtrinsics(stereoParams);
```

The calculation of parallax and the display of depth have been explained in detail in our second task, and we will not go into details here, but the prerequisite for generating 3D point clouds is also to obtain the corresponding parallax map obtained in the previous step. We will focus on the generation part of the 3D point cloud.

We first used reconstructScene to calculate three-dimensional point clouds according to parallax map and reprojection matrix, filtered invalid points with depth greater than 10 meters, converted coordinate units from millimeters to meters, and finally generated point cloud objects through pointCloud combined with color information.

```
% Reconstruct the 3D scene based on the disparity map and reprojection matrix
points3D = reconstructScene(disparityMap, reprojectionMatrix);

% Filter out invalid 3D points by removing points with a z-coordinate greater than 10 meters
validPoints = points3D(:, :, 3) < 10; % z-coordinate represents the depth (distance from the
camera)
points3D(validPoints) = NaN; % Set invalid points to NaN to exclude them from the point
cloud

% Convert the 3D points' units from millimeters to meters
points3D = points3D ./ 1000;

% Create a point cloud object using the 3D points and the color data from the rectified left
image
ptCloud = pointCloud(points3D, 'Color', frameLeftRect);
```

After that, we save the generated point cloud data as a.ply file, which supports ASCII encoding for subsequent analysis and visualization.

```
% save the 3D cloud points
pcwrite(ptCloud,'result.ply','Encoding','ascii');
```

Create a 3D point cloud viewer through pcplayer, set the visual scope and axis orientation, and use view to load the generated point cloud data into the viewer for display.

```
% Create a 3D point cloud viewer
player3D = pcplayer([-3, 3], [-3, 3], [0, 3], 'VerticalAxis', 'y', ...
    'VerticalAxisDir', 'down');

% Display the generated point cloud in the viewer
view(player3D, ptCloud);
```

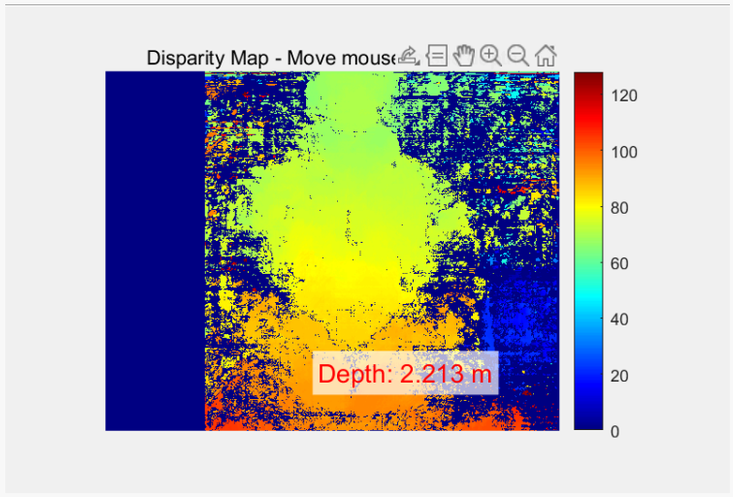
In this way, the effect of constructing a 3D point cloud with a certain frame is successfully completed.

6.3. Experiment Result

In PointCloudVisualization. M file test under the same directory - left folders and test - right folder, respectively, at the same time put on one piece of the binocular cameras do two photographs, pay attention to take the photos of clarity and completeness. Run the code to get the final result.

This is the original image taken by the binocular camera:

The resulting parallax map and 3D point cloud map are as follows:



We found that the effect of the finally constructed parallax map and 3D point cloud map is quite good, that is, some points in the 3D point cloud map are not displayed, and they are black, which may have the following reasons:

1. Points with depth values greater than 10 meters are filtered out. As a result, these points are marked as invalid, resulting in not being visible in the point cloud map. Indeed, when we took these two pairs of photos, the distant wall behind us may be more than 10 meters away from the camera.
2. The camera we buy is not a professional camera, (cheap), with limited accuracy, low resolution, and limited details captured per pixel.

However, in general, most of the contours of the constructed 3D point cloud images can be clearly identified.

At this point, the third task of constructing a 3D point cloud based on a captured frame is completed.

7. Reconstruct the Full 3D Model

In the last step of the multi-view 3D model reconstruction process, we first obtained the multi-view photos of the busts of the team members, and then generated the multi-view point cloud through the code that generated the point cloud. Next, the noise is removed by color denoising, distance denoising and so on. Finally, the point cloud of all perspectives is combined by coarse registration and fine registration to obtain the final 3D model. In this process, because the accuracy of the final point cloud data obtained in the process of de-noising and matching is limited, we use CloudCompare tool to make the accuracy of de-noising and matching higher. The following is the project structure for this section:

```
MultiView3DReconstruction
    ├── MultiViewStereo
    |   ├── left
    |   ├── right
    |   └── ContinuousPhotography.py
    ├── CreatePlys
    |   ├── left
    |   ├── right
    |   ├── ply
    |   ├── SeeClouds.m
    |   └── create_ply.m
    ├── FilterByColor
    |   ├── filter
    |   ├── ply
    |   ├── SeeClouds.m
    |   └── FilterByDistance.m
    ├── FilterByDistance
    |   ├── afterColorFilter
    |   ├── final
    |   ├── SeeClouds.m
    |   ├── ThreeInitial.m
    |   ├── mergedInitial.ply
    |   └── FilterByDistance.ipynb
    └── FinalPresentation
    |   ├── SeeClouds.m
    |   ├── final.mp4
    └── └── final.ply
```

7.1. Correlation Principle

7.1.1. Point Cloud Registration

The goal of point cloud registration is to align point clouds from multiple perspectives into the same coordinate system to form a complete 3D model. There are two common point cloud registration methods: coarse registration and fine registration.

- **Initial Alignment**

The goal of rough registration is to quickly align the point clouds roughly and reduce rotation and displacement errors between the two points. The common method is ICP iterative optimization algorithm.

The goal of ICP algorithm is to find the rigid transformation between source point cloud P and target point cloud Q
T (including rotation R and shift t) to minimize the error. Have a calculation formula:

$$T = \arg\min_{R,t} \sum_{p_i \in P} \|Rp_i + t - q_i\|^2$$

Where p_i is the point in point cloud P, and q_i is the closest point to p_i in point cloud Q. For each point in P, find the nearest neighbor in Q. Calculate the rigid transformation T. Update the location of $P \leftarrow T(P)$. Iterate until the errors converge.

At the same time, feature matching is also necessary to extract the feature points of the point cloud and establish a matching relationship between the feature points for preliminary alignment:

Extract key points and descriptors using $SIFT$ or ORB. Find point clouds based on nearest neighbor matching descriptors Preliminary matching pairs of P and Q. The initial rigid transformation T is estimated by removing the mismatched point pairs by RANSAC method.

- **Fine Alignment**

On the basis of coarse registration, fine registration further refines the alignment process of Point clouds. Common fine registration methods include Weighted ICP and point-to-plane ICP. These methods improve alignment accuracy by weighting the point pairs in the point cloud and optimizing the rigid transformation (rotation matrix and displacement vector).

1. Weighted ICP

Weighted ICP is an extension of traditional ICP that exerts different influence on the alignment process of different pairs by assigning a weight w_i to each point pair. The weight w_i is usually assigned according to the distance of the point pair or the consistency of the normal vector.

$$T = \arg\min_{R,t} \sum_{p_i \in P} w_i \|Rp_i + t - q_i\|^2$$

Where: $T = \{R, t\}$ is the rigid transformation we want to optimize, including the rotation matrix R and the displacement vector$t$. p_i is the point in the source point cloud and q_i is the corresponding point in the target point cloud. w_i is the weight of each point pair and determines how much that point pair contributes to the optimization result.

The weight w_i can be assigned based on the distance of the point pairs or the normal vector consistency between the point pairs. For example, points that are far from the viewpoint or poorly matched can be given less weight, thus reducing their impact on the overall registration.

2. Point-to-Plane ICP

The error measurement method of point-to-plane ICP is different from that of traditional point-to-point ICP. Traditional point-to-point ICP calculates the Euclidean distance between the source and target points, while point-to-plane ICP calculates the point-to-plane distance, taking into account the normal vector of the point cloud surface, thereby improving alignment accuracy and convergence speed.

$$T = \arg\min_{R,t} \sum_{p_i \in P} (n_i^T (Rp_i + t - q_i))^2$$

Where: n_i is the normal vector of the target point cloud q_i. $Rp_i + t$ is the coordinate of the source point cloud p_i after rotation and displacement. $n_i^T (Rp_i + t - q_i)$ represents the distance from the point p_i to the plane, where the normal vector of the plane is n_i.

7.1.2. Point Cloud Fusion and Optimization

The goal of point cloud fusion is to integrate multiple aligned point clouds into a complete model while optimizing redundant data and noise.

- **Redundant Point Removal**

The multi-view point cloud has a large number of redundant points in the overlapping region, which can be simplified by voxel filtering. Divide the space into cube grids (voxels).In each voxel, only one representative point (usually a central or average point) is retained.

$$p_{\text{voxel}} = \frac{1}{N} \sum_{i=1}^{N} p_i$$

Where p_i is the points within the voxel and N is the number of points within the voxel.

- **Noise Filtering**

Detect isolated points (noise points) in the point cloud and remove outliers by analyzing the neighborhood distance of the points.
Formula: Calculate the mean neighborhood distance of point p:

$$d_{\text{mean}}(p) = \frac{1}{k} \sum_{i=1}^{k} \|p - p_i\|$$

If $d_{mean}(p)$ is greater than a certain threshold, it is considered an anomaly.

In our project, noise has a great impact on our final experimental results. We use a variety of methods for denoising, including manual rough denoising and algorithmic precise denoising, etc., which will be described in detail below.

7.2. Practice Process

- **Get multi-view point clouds**

When it comes to 3D reconstruction of an object, getting photos from different angles is a crucial step. In order to ensure the accuracy and integrity of the reconstruction, when shooting, try to maintain a uniform Angle interval between the camera and the object to avoid excessive overlap or blind areas in local areas. Ensure the stability of the camera during the shooting process to avoid camera shake or offset. Therefore, we wrote a script to get a picture by taking a series of photos.

```
# ContinuousPhotography.py
import cv2
import os
from time import sleep

# Function to initialize the cameras
def initialize_cameras(left_camera_id=0, right_camera_id=1):
    left_cap = cv2.VideoCapture(left_camera_id)
    right_cap = cv2.VideoCapture(right_camera_id)

    if not left_cap.isOpened() or not right_cap.isOpened():
        print("Unable to open cameras. Please check the connection.")
        return None, None

    return left_cap, right_cap

# Function to create directories for saving images
def create_folders(left_folder='left', right_folder='right'):
    os.makedirs(left_folder, exist_ok=True)
    os.makedirs(right_folder, exist_ok=True)

# Function to capture and save images
def capture_and_save_images(left_cap, right_cap, left_folder, right_folder, image_count=50):
    for i in range(image_count):
        # Capture frames from both cameras
        ret_left, frame_left = left_cap.read()
        ret_right, frame_right = right_cap.read()

        if not ret_left or not ret_right:
            print("Unable to capture frames. Please check the cameras.")
```

```
            break

        # Save the images
        left_image_path = os.path.join(left_folder, f"left_{i+1}.jpg")
        right_image_path = os.path.join(right_folder, f"right_{i+1}.jpg")
        cv2.imwrite(left_image_path, frame_left)
        cv2.imwrite(right_image_path, frame_right)

        # Output the saved file paths
        print(f"Saved Left Image: {left_image_path}, Right Image: {right_image_path}")

        # Display images
        cv2.imshow('Left Camera', frame_left)
        cv2.imshow('Right Camera', frame_right)

        sleep(5)

# Function to release the cameras and close all windows
def release_resources(left_cap, right_cap):
    left_cap.release()
    right_cap.release()
    cv2.destroyAllWindows()

# Main function
def main():
    # Initialize cameras
    left_cap, right_cap = initialize_cameras()

    if left_cap is None or right_cap is None:
        return

    # Create folders to save images
    create_folders(left_folder='left', right_folder='right')

    # Capture and save 50 pairs of images
    capture_and_save_images(left_cap, right_cap, left_folder='left', right_folder='right',
image_count=50)

    # Release resources after capturing images
    release_resources(left_cap, right_cap)

    print("Image capturing and saving completed.")

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

The next is to generate a point cloud image of each direction, the specific method is similar to generating a point cloud image of a perspective, but by traversing all pairs of pictures under the photo folder to generate different point clouds specific methods will not be described.

```
% Part of create_ply.m
% loop through 1 to 24 image pairs
for i = 1:24
    % build image file name
    leftImgFile = fullfile(leftImagePath, sprintf('left_%d.jpg', i));
```

```
    rightImgFile = fullfile(rightImagePath, sprintf('right_%d.jpg', i));

    % read images
    leftImg = imread(leftImgFile);
    rightImg = imread(rightImgFile);

    % rectify images based on stereoParams
    [frameLeftRect, frameRightRect, reprojectionMatrix] = rectifyStereoImages(leftImg,
rightImg, stereoParams);

    % convert to grayscale
    frameLeftGray  = im2gray(frameLeftRect);
    frameRightGray = im2gray(frameRightRect);

    % apply guided filter
    frameLeftGray = imguidedfilter(frameLeftGray, 'DegreeOfSmoothing', 0.05);
    frameRightGray = imguidedfilter(frameRightGray, 'DegreeOfSmoothing', 0.05);

    % calculate disparity map
    disparityMap = disparitySGM(frameLeftGray, frameRightGray, 'DisparityRange', [0 128],
'UniquenessThreshold', 10);

    % calculate 3D points based on disparity map and reprojectionMatrix
    points3D = reconstructScene(disparityMap, reprojectionMatrix);

    % filter invalid points
    validPoints = points3D(:, :, 3) < 10; % assume z coordinate greater than 10 meters is
invalid
    points3D(validPoints) = NaN;

    % convert physical unit from millimeters to meters
    points3D = points3D ./ 1000;

    % create point cloud object
    ptCloud = pointCloud(points3D, 'Color', frameLeftRect);

    % save point cloud to local disk
    plyFileName = fullfile(plyOutputPath, sprintf('result_%d.ply', i));
    pcwrite(ptCloud, plyFileName, 'Encoding', 'ascii');

    % print progress information
    fprintf('Processed pair %d and saved as %s\n', i, plyFileName);
end
```

- **Filter invalid points and noise**

Ok, now we can look at the point cloud scenario using the following code:

```
% read ply file
filename = 'your_file.ply'; % replace with your ply file path
ptCloud = pcread(filename);

% create a point cloud viewer
% define the range and direction of the x, y, z axes
player3D = pcplayer([-3, 3], [-3, 3], [0, 3], ...
```

```
    'VerticalAxis', 'y', ...
    'VerticalAxisDir', 'down');

% view the generated 3D point cloud
view(player3D, ptCloud);

% add color information
if isfield(ptCloud, 'Color')
    player3D.Color = ptCloud.Color;
end

% keep the graph window open until the user closes it
drawnow limitrate; % ensure the graph interface is updated immediately
disp('Press any key to close the viewer...');
pause;

% pause the program, wait for user interaction
```

We can find that there are not only the portraits we need in the point cloud, but also many unnecessary point clouds caused by walls and light, which will affect the quality of the final 3D model, because we do not need these chaotic point clouds in the process of point cloud registration, so we need to carry out invalid point elimination.

Invalid point elimination, that is, to delete the point cloud of the wall. Here, we found a huge difference between the color of the wall and the color of our portrait, so we used the color information cull to remove these invalid points. Here is the relevant code.

```
% define the input and output folder path
inputFolder = 'ply';  % the folder of the input ply files
outputFolder = 'filter';  % the folder of the output filtered ply files

% ensure the output folder exists
if ~exist(outputFolder, 'dir')
    mkdir(outputFolder);
end

% traverse all the files to be processed from result_1.ply to result_24.ply
for i = 1:24
    inputFile = fullfile(inputFolder, sprintf('result_%d.ply', i));
    outputFile = fullfile(outputFolder, sprintf('filtered_result_%d.ply', i));

    % check if the input file exists
    if exist(inputFile, 'file')
        % filter and save the point cloud
        filter_ply_file1(inputFile, outputFile);
    else
        warning(['Input file does not exist: ', inputFile]);
    end
end

disp('Batch filtering completed.');

% define the filtering function
function filter_ply_file1(inputFile, outputFile)
    % read the ply file
```

```
    ptCloud = pcread(inputFile);

    % get the color information of the point cloud
    colors = ptCloud.Color;

    % define the color range
    lowerBound = 170;
    upperBound = 220;

    % find the indices of the points whose RGB values are within the specified range
    inRangeIdx = all((colors >= lowerBound) & (colors <= upperBound), 2);

    % delete the points whose RGB values are within the specified range
    filteredLocations = ptCloud.Location(~inRangeIdx, :);
    filteredColors = colors(~inRangeIdx, :);

    % create a new point cloud object
    filteredPtCloud = pointCloud(filteredLocations, 'Color', filteredColors);

    % save the filtered point cloud to a new ply file
    pcwrite(filteredPtCloud, outputFile, 'Encoding', 'ascii');

    disp(['Filtered PLY file saved to: ', outputFile]);
end
```

After observation, we found that the points in the point cloud range from 170 to 220 are invalid points (non-human image points) that we need to eliminate. So we get the color information of the point cloud and filter the point cloud data according to the specified color range (170 to 220).

After the point cloud color filtering, we found that the distance of some point clouds is not appropriate, these point clouds are either too far away or too close, which is caused by the error when generating 3D point clouds, here we need to remove these points.

```
from plyfile import PlyData, PlyElement
import numpy as np

def filter_ply_by_z(file_path, output_path, z_min=2, z_max=3):
    # read ply file
    plydata = PlyData.read(file_path)

    # get vertex element and property names
    vertex_element = plydata.elements[0]
    property_names = vertex_element.data.dtype.names

    # convert vertex data to numpy array, and add other properties (such as color)
    vertex_data = np.array(
        [tuple(row) for row in vertex_element.data],
        dtype=np.dtype([(name, vertex_element.data.dtype[name]) for name in property_names])
    )

    # filter out points with z coordinate between z_min and z_max
    mask = (vertex_data['z'] >= z_min) & (vertex_data['z'] <= z_max)
    filtered_vertex_data = vertex_data[mask]

    # create new vertex element, keep original properties
```

```
    new_vertex_element = PlyElement.describe(filtered_vertex_data, 'vertex')

    # build new PLY data structure and keep original file's text mode
    new_ply_data = PlyData([new_vertex_element], text=plydata.text)

    # write to new PLY file
    new_ply_data.write(output_path)

 # loop through files
for i in range(1, 25):
    input_file = f'afterColorFilter/filtered_result_{i}.ply'  # input PLY file path
    output_file = f'final/final_{i}.ply'  # output PLY file path
    filter_ply_by_z(input_file, output_file)
    print(f"Processed file {i}")
print("All files processed")
```

We need to filter the point cloud data in the PLY file. Most of the normal point cloud ranges we look at here are directly on the Z-axis of 2-3. We first read the PLY file, extract the vertex elements and their attributes, and then convert the vertex data into a NumPy array. Next, the code filters the point cloud data according to the range of Z coordinates, creates new vertex elements, builds a new PLY data structure, and finally writes the filtered data to a new PLY file.

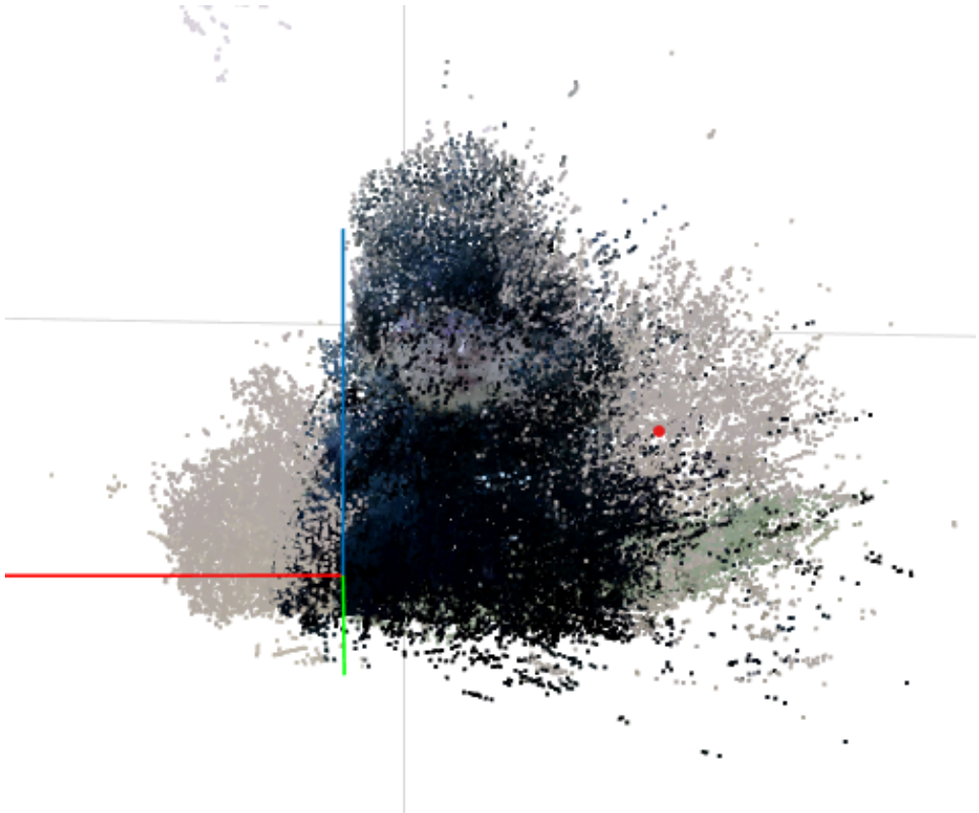
- **Preliminary effects of 3D reconstruction**

Now that we have the point cloud map in all directions after our own de-noising and de-invalidation, we are going to merge it into a point cloud. Through the rough registration and fine registration mentioned above, we can initially construct the effect of 3D reconstructed objects.

```
% ThreeInitial.m
% read point cloud
ptClouds = cell(1, 24); % to store all point clouds
for i = 1:24
    ptClouds{i} = pcread(sprintf('final/final_%d.ply', i)); % read each point cloud
end

% Initial Alignment
% select the first point cloud as the base
basePtCloud = ptClouds{1};

for i = 2:24
    % apply ICP coarse registration to each point cloud
    ptCloudTransformed = ptClouds{i};
    [tform, ptCloudAligned] = pcregistericp(ptCloudTransformed, basePtCloud,
'MaxIterations', 50, 'Tolerance', [0.001, 0.001], 'Metric', 'pointToPoint');
    ptClouds{i} = ptCloudAligned;  % update to the aligned point cloud
end

% Fine Alignment
% on the basis of coarse registration, use more fine ICP refinement
for i = 2:24
    ptCloudTransformed = ptClouds{i};
    [tform, ptCloudAligned] = pcregistericp(ptCloudTransformed, basePtCloud,
'MaxIterations', 100, 'Tolerance', [0.0001, 0.0001], 'Metric', 'pointToPlane');
    ptClouds{i} = ptCloudAligned;
end
```

```
% Point Cloud Fusion
% merge all aligned point clouds
mergedPtCloud = ptClouds{1};
for i = 2:24
    mergedPtCloud = pcmerge(mergedPtCloud, ptClouds{i}, 0.01); % 0.01 is the merging
distance threshold
end

% Post-Processing
% use voxel filtering to reduce the density of the point cloud
gridSize = 0.005; % set the voxel size
mergedPtCloud = pcdownsample(mergedPtCloud, 'gridAverage', gridSize);

% visualize the merged point cloud
player3D = pcplayer([-3, 3], [-3, 3], [0, 3], 'VerticalAxis', 'y', 'VerticalAxisDir',
'down');
view(player3D, mergedPtCloud);

% save the final point cloud
pcwrite(mergedPtCloud, 'mergedInitial.ply');
```

First, we read 24 point cloud files, then do coarse alignment and fine alignment, and finally merge all the aligned point clouds and do voxel filtering to reduce the density of the point cloud. Finally, it visualizes the merged point cloud and saves it as a file. The final result is shown below:



We can see that the 3D reconstructed point cloud after processing the above steps, although we can see that we build a new person, but the effect is not particularly ideal, so I named this step here "ThreeInitial", this is the initial version of the 3D reconstruction model, we can observe the following reasons:

1. Coarse registration accuracy is not high:
   In the initial registration process of the point cloud, a simple ICP algorithm is used, which is helpful for the initial alignment, but the retention of details and accuracy may not be high enough. Because ICP algorithms rely on point-to-point matching, they may not be able to accurately capture details between all point clouds, especially if there is noise or occlusion between point clouds.
2. Point cloud noise and redundant data:
   Indeed, we can see that even after two steps of de-noising, the resulting point cloud still has a lot of noise in the part near the portrait. Point cloud data often contains noise, redundant points, and non-essential data that does not belong to the target object. Even if redundancy point removal and noise filtering are performed during the registration process, some details may still be lost or mismatched. This is because the collection process of the point cloud itself inevitably brings some errors.

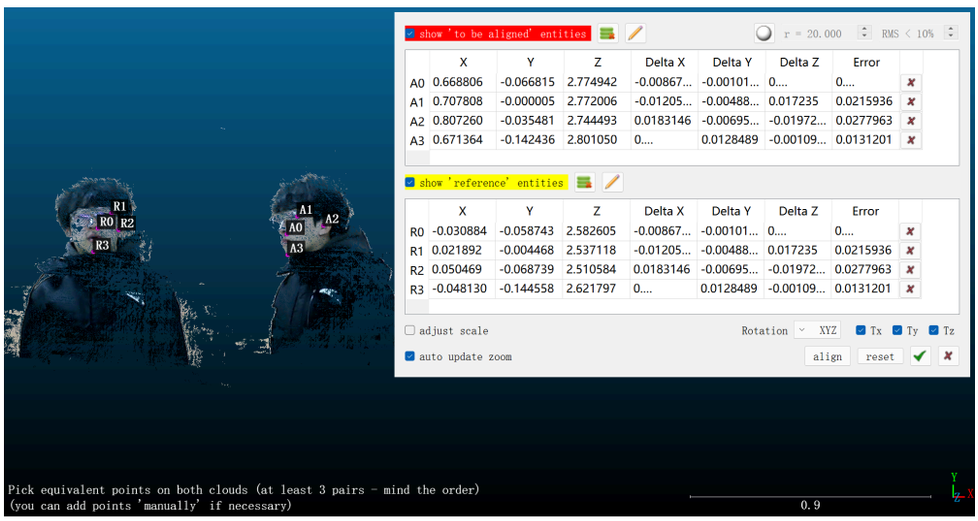
Therefore, we are going to use a more powerful basis for denoising operations and point cloud registration. We introduce another approach below: 3D reconstruction based on CloudCompare assistive tools.

- **3D reconstruction based on CloudCompare assistive tool**

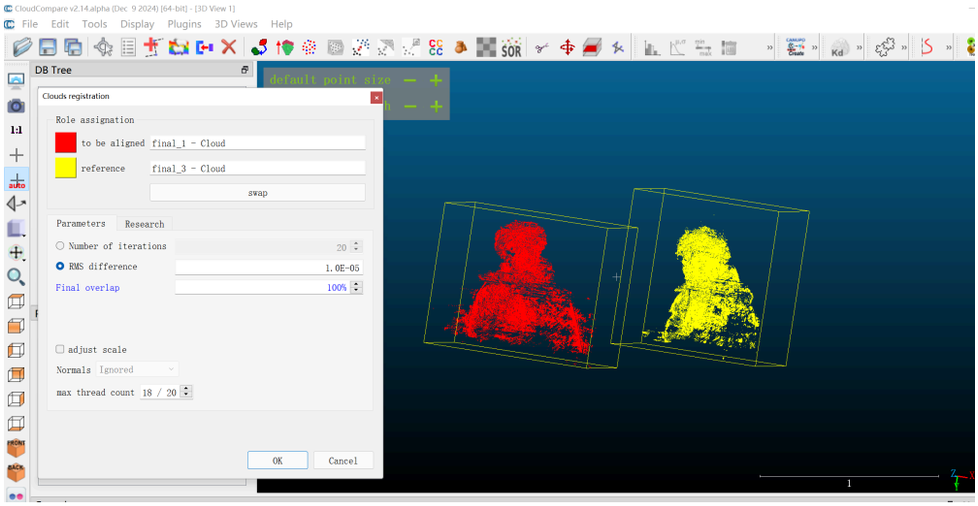
"ThreeInitial" means that the current rebuild model belongs to a preliminary, base rebuild version. Although the preliminary point cloud registration and fusion model can basically restore the shape of the object, due to the registration accuracy, noise effects and algorithm limitations, the details may not be fully preserved, resulting in unsatisfactory accuracy and visualization effect. To solve this problem, we found a point cloud registration assistance tool "CloudCompare".

In point cloud processing software such as CloudCompare, it is indeed possible to manually remove noise and unnecessary point cloud data. Select Tools > Scalar Field > Filter by value in CloudCompare to select points with values within a certain range and remove them. In this way, we can further denoise the point cloud filtered by the above program.

Next, we can perform point cloud merge matching. In CloudCompare, we can manually select the matching points of two point clouds and then merge them. The manual key-point selection and rotation matching methods in CloudCompare combine feature matching and rigid registration algorithms. As shown below, we can manually mark key points. Then in the process of matching, the system will automatically calculate the rotation change matrix.



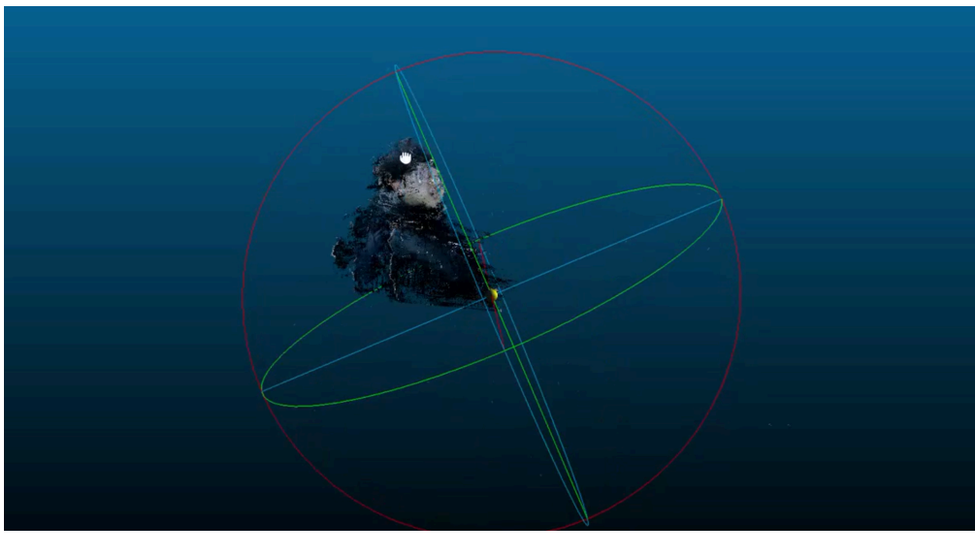
Of course, we can also use the intelligent fine registration method in CloudCompare to merge the point cloud, which automatically calculates the best registration transformation between two or more point clouds through the ICP algorithm and other optimization strategies. Reduce manual operation and improve the accuracy of high point cloud alignment.



After that, we can perform simple operations such as de-overlapping, save the non-PLY file and export it, so that we can further improve the accuracy of the 3D model based on the multi-view point cloud we obtained after the above coarse filtering.

7.3. Experiment Result

In the last step of the multi-view 3D model reconstruction process, we first obtained the multi-view photos of the busts of the team members, and then generated the multi-view point cloud through the code that generated the point cloud. Next, the noise is removed by color denoising, distance denoising and so on. Finally, the point cloud of all perspectives is combined by coarse registration and fine registration to obtain the final 3D model. In this process, because the accuracy of the final point cloud data obtained in the process of de-noising and matching is limited, we use CloudCompare tool to make the accuracy of de-noising and matching higher. This is our final 3D model, which can also be seen in final.mp4 in the FinalPresentation folder.



We can see that the 3D model finally constructed is relatively clear, and the contours and parts of people can be distinguished. There are still some minor flaws, which should be the reason why the camera itself is too low in accuracy.

8. The Summary

Although the 3D model was successfully generated through the above steps, from the actual effect, the quality of the final generated 3D model is not particularly high, should be limited by the camera resolution, due to the use of the camera resolution of 480p, the details and accuracy of the image are limited. Low-resolution images do not provide enough detail, especially in parts far from the camera, which can easily lead to poor accuracy of the parallax map, which affects the quality of the point cloud. However, on the whole, the reconstructed object

shape is clearly visible and has good coherence in multiple perspectives. The final 3D reconstruction is already quite good.

In summary, this project shows the whole process from camera calibration, image acquisition, depth display, point cloud generation, registration, and 3D model reconstruction. Although the model has some flaws in some details, it also provides us with the direction to further optimize and improve the accuracy of the model. In the future, we can improve the quality of reconstructed models by increasing camera resolution, optimizing point cloud processing algorithms, and adding more data from more perspectives.

By solving specific problems, I not only improved my programming skills, but also learned how to debug and optimize models and enhanced my problem-solving skills. In addition, the teamwork in the project also benefited me a lot. Through communication and collaboration with team members, I learned how to divide the labor and integrate different skills and resources, which improved my teamwork ability. This project not only allowed me to master more techniques, but also exercised my ability to deal with challenges in practice, and accumulated valuable experience for future study.