Master Project

Bin Picking Project

Puneet Singh Arora

Immatriculation Number :- 1056132

Examiner: Prof. Micheal Wagner
Date of submission: 07/24/2024

Formale Erklärung der Urheberschaft

Ich, der/die Unterzeichnete, erkläre hiermit förmlich, dass ich der/die alleinige Verfasser/in dieses Projektberichts bin und den gesamten Inhalt unabhängig verfasst habe. Ich habe diesen Bericht oder Teile davon noch nie einer anderen Institution zur Begutachtung vorgelegt. Darüber hinaus habe ich keine anderen als die im Bericht ausdrücklich angegebenen Quellen und Hilfsmittel benutzt. Alle direkten und indirekten Zitate sind deutlich als solche gekennzeichnet.

Datum: 24. Juli 2024

Ort: Rosenheim

Puneet Singh Arora

Abstract

In this Project a a solution to the complex problem of robotic bin picking in unstructured environments was developed, with a key emphasis on utilizing open-source software tools. A meticulously calibrated stereo camera system was employed to capture high-resolution images of the bin's contents from multiple perspectives. Dense and accurate depth maps, enabling faithful 3D reconstruction of the scene even in the presence of clutter and occlusions, were generated through analysis of the disparities between left and right images. Data analysis and visualization for this project were conducted using Python 3.11 leveraging the open-source libraries OpenCV, NumPy, and Matplotlib also the open source codes from Allied Vision which include important library functions for the proper functioning of cameras like shutter closing and opening ,white balance, shutter timing .

Recognizing the challenges posed by overlapping objects, varying lighting conditions, and complex geometries, a novel algorithm tailored specifically for bin picking applications was devised.

The system's performance was evaluated through testing in a simulated scenario, encompassing a range of object types, sizes, materials, and bin configurations. Results consistently demonstrated the system's robustness, accuracy, and adaptability, even in challenging conditions such as varying illumination, reflective surfaces, and complex object geometries. This project constitutes a significant advancement in the field of robotic bin picking, paving the way for wider adoption of automation in manufacturing, logistics, and other industries where the handling of unstructured objects is a critical bottleneck. By overcoming the limitations of traditional approaches, this stereo vision-based system offers a practical and scalable solution for automating bin picking tasks, leading to enhanced efficiency, reduced costs, and improved safety in industrial and logistical processes.

In conclusion, this project demonstrates the successful development and implementation of a robust stereo vision-based system for robotic bin picking, paving the way for advancements in automation and grasp planning strategies in industrial and logistical processes.

Table of Contents

Contents

1 Introduction

Bin picking, a persistent challenge in automation and robotics, necessitates innovative solutions due to the inherent variability of bin environments. Traditional approaches, often relying on meticulously controlled environments or specialized grippers, limit flexibility and adaptability to unstructured scenarios. Stereo vision, however, offers a promising alternative by mimicking human depth perception through images captured from slightly offset viewpoints. By analyzing the disparity between these images, stereo vision systems generate accurate 3D representations of bin contents, enabling robots to precisely perceive object location and orientation. This depth information is crucial for tasks like object recognition, pose estimation, and grasp planning

.

This project explores the development and implementation of a stereo vision-based system for robotic bin picking, aiming to enhance efficiency and accuracy in real-world, unstructured environments.

By leveraging stereo vision's ability to provide detailed 3D information, the system enables robust object localization, precise pose estimation, and reliable grasp planning, overcoming the limitations of traditional approaches that struggle with object variability and occlusion. This work paves the way for advancements in automation and manipulation strategies in industries such as manufacturing, logistics, and e-commerce fulfillment, where the ability to efficiently handle objects in diverse configurations is paramount. The successful implementation of this system not only addresses a persistent challenge in robotics but also opens doors for further projects into more sophisticated object manipulation and autonomous decision-making in unstructured environments.

Objective

The overarching objective of this undertaking is to conceptualize and assess a robust stereo vision system capable of delivering precise three-dimensional reconstruction of objects within unstructured bin environments. Such environments encompass industrial parts bins, warehouse order picking bins, and recycling containers.

The realization of this vision entails the calibration of a stereo camera setup utilizing a chessboard pattern. Subsequently, corresponding points within captured images of the cluttered bin contents are identified and matched, culminating in the generation of accurate depth maps. The system endeavors to provide reliable three-dimensional object pose estimations, thus facilitating subsequent robotic tasks such as grasp planning and manipulation. This, in turn, enhances automation and efficiency within diverse industrial and logistical settings.

Methodology

The selection of software instruments is a pivotal element in any project, as it can have a significant impact on the expenses, ease of use, and long-term support of the project. In this study, a deliberate decision was made to utilize open-source tools and libraries for the development and evaluation of the stereo vision system. This decision was primarily driven by multiple key considerations.

Firstly, open-source tools and libraries offer substantial cost advantages. Unlike proprietary software, open-source tools are freely available and do not necessitate licensing fees, making them a cost-effective option for the project.

Secondly, open-source tools and libraries provide greater accessibility and flexibility. With open-source tools, researchers possess the freedom to modify and adapt the code to meet their specific needs and requirements. This flexibility helped to customize the tools and libraries to seamlessly integrate with the stereo vision system setup.

Thirdly, open-source tools and libraries frequently come with a large and active community of developers and users. This community provides valuable support, documentation, and troubleshooting resources. The research team benefited from this community support, as they were able to swiftly resolve technical issues and learn from the experiences of others who had employed the same tools and libraries.

Among the open-source tools and libraries employed in this study, Python 3.11 emerged as the primary development platform. Python is a versatile and widely utilized programming language that offers numerous advantages for scientific research. It is renowned for its simplicity of use, readability, and extensive library support.

The selection of Python was further supported by the availability of robust open-source libraries such as OpenCV, NumPy, and Matplotlib. OpenCV is a comprehensive library for computer vision and image processing, providing a broad range of functions for tasks such as feature detection, object tracking, and image manipulation. NumPy is a fundamental library for scientific computing, offering efficient data structures and functions for numerical operations. Matplotlib is a versatile library for data visualization, enabling the creation of various plots and charts.

In addition, the Vimba Viewer application was employed for its compatibility with the Allied Vision cameras utilized in this project, streamlining the image acquisition and camera control process. By utilizing these open-source tools, this project not only ensures accessibility and cost-effectiveness but also benefits from the continuous development and support provided by their dedicated communities.

Sequence of Operations:

- **Vimba-based Dual Camera System for Real-Time Image Capture and Display**

This Python script provides a robust solution for capturing, processing, and displaying real-time images from two Vimba-compatible cameras. The system is designed to be interactive, allowing users to save images on-demand, and features a software-triggered acquisition mechanism for consistent results.

The primary objectives of this script are:

- Real-Time Image Acquisition: Capture images simultaneously from two Vimba cameras in a continuous stream.
- Image Processing: Perform basic image processing operations, including:
    - Grayscale conversion for simplified analysis or reduced data size.
    - Resizing to a standard 640x480 resolution for display and consistency.
- Live Display: Present the captured images in separate windows for real-time monitoring and visual feedback.
- On-Demand Saving: Enable users to save individual image pairs to disk with a single key press.

Technical Overview

1. Camera Discovery and Initialization: The script identifies connected Vimba cameras using the vmbpy SDK, configures camera settings like auto-exposure (where available), and sets up a software trigger for image acquisition.
2. Error Handling: The script includes a check for the initial detection of cameras, it's beneficial to implement more comprehensive error handling for real-world scenarios.
3. Image Acquisition and Processing: The core functionality resides in the custom Handler class, which efficiently receives image frames from the cameras, converts them to the appropriate format, and queues them for display. The main loop of the script handles image resizing, grayscale conversion, and the display of images in separate windows labeled "Right Camera" and "Left Camera".
4. User Interaction: The script constantly monitors for user input. Pressing the ‘Esc’ key gracefully terminates the streaming process and closes the display windows, while pressing the ’s’ key triggers the saving of the current image frames from both cameras.

Libraries and Dependencies

- vmbpy (Vimba Python SDK): Essential for communication and control of Vimba-compatible machine vision cameras. Ensure that the Vimba SDK is installed and properly configured on your system.
- cv2 (OpenCV): A versatile computer vision library used for image processing, analysis, and display.

- numpy: A fundamental library for numerical operations, providing the foundation for efficient image manipulation within OpenCV.
- time, queue: Additional libraries used for time management and frame queuing to ensure smooth image streaming.

Installation and Configuration

1. Vimba SDK: Download and install the Vimba SDK from the official Allied Vision website (camera manufacturer's site). Follow the provided instructions for your specific operating system and camera model.
2. Python Libraries: Use the pip package manager to install the required Python libraries:
3. Camera Connection: Vimba cameras are properly connected to the computer via . Verify that the cameras are detected by the Vimba SDK.
4. Camera Settings: Consider manually adjusting the exposure settings within the script to optimize image quality for your specific lighting conditions. Refer to your camera's documentation for detailed instructions.

Usage :

1. The live camera feeds will be displayed in separate windows.
2. Press s to save the current image pair to disk.
3. Press Esc to exit the script.

Code Explanation:

The code Imports necessary libraries, including:

- `vmbpy` for Vimba camera access
- `cv2` (OpenCV) for image processing and display
- `numpy` for numerical operations
- Custom modules `Calibration`, `disparity`, and `disparityGUI2`

```
import cv2 as cv
import numpy as np
from vmbpy import *
import time
from queue import Queue
from typing import Optional
from typing import Callable
import matplotlib.pyplot as plt
import Calibaration
import Disparity
import DISPARITY as disp
```

onMouse Function:

This function is designed to be a callback for mouse events on an image display.

Functionality:

- If the left mouse button is clicked (cv.EVENT_LBUTTONDOWN) on the image display of the disparity map, it will look up the disparity value disparityNormalized[y][x] at the clicked location. This value represents the estimated distance from the camera in centimeters to the 3D point corresponding to that pixel in the image.
- The disparity value is then printed to the console in the format "Distance in Centimeters: {distance}".

- Finally, the function returns the distance value.

```
def onMouse(event, x,y, flag, disparityNormalized):
    if event == cv.EVENT_LBUTTONDOWN:
        distance = disparityNormalized[y][x]
        print("Distance in Centeimeters{}".format(distance))
        return distance
```

Camera Setup Functions (get_camera, setup_camera):

get_camera() Function:

- Uses a context manager (with) to ensure proper resource management with the Vimba system.
- Retrieves a list of all connected Vimba cameras using vmb.get_all_cameras().
- Checks if any cameras are found. If not, it prints an error message and exits the script.
- Returns the list of discovered cameras.

setup_camera(cam) Function:

- Takes a Camera object as input.
- Uses a context manager (with) for the camera, ensuring proper setup and cleanup.
- Tries to set auto-exposure and auto-white balance to 'Continuous'. If these features aren't supported by the camera, it gracefully skips them.
- Tries to get the first stream (stream = cam.get_stream()[0]) and adjust the packet size for optimal data transmission using stream.GVSPAdjustPacketSize.run(). It waits until the adjustment is complete.

```
opencv_display_format = PixelFormat.Bgr8

def get_camera():
        with VmbSystem.get_instance() as vmb:
                cameras = vmb.get_all_cameras()
                if  not cameras:
                        print("No cameras Detected")
                        exit()

        return cameras

def setup_camera(cam: Camera):
        with cam:

                try:
                        cam.ExposureAuto.set('Continuous')
                except(AttributeError, VmbFeatureError):
                        pass

                try:
                        cam.BalanceWhiteAuto('Continuous')
                except(AttributeError, VmbFeatureError):
                        pass
                try:
                        stream = cam.get_stream()[0]
                        stream.GVSPAdjustPacketSize.run()
                        while not stream.GVSPAdjustPacketSize.is_done():
                                pass
                except(AttributeError, VmbFeatureError):
                        pass
```

Handler Class:

- Defines a custom class to handle incoming image frames from the cameras.
- __init__: Initializes a queue (display_queue) to hold up to 10 images.
- get_image: Provides a method to retrieve an image from the queue (blocking if the queue is empty).
- __call__: The main callback function executed when a new frame arrives.
    - Checks if the frame is complete.
    - If the frame format isn't already suitable for OpenCV, it converts it.
    - Adds the OpenCV-formatted image to the queue.
    - Re-queues the frame buffer to be reused for the next image.

```
class Handler:
        def __init__(self):
                self.display_queue = Queue(10)

        def get_image(self):
                return self.display_queue.get(True)

        def __call__(self, cam:Camera, stream:Stream, frame:Frame):

                if frame.get_status() == FrameStatus.Complete:
                        print('{} acquired {}'.format(cam, frame), flush = True)

                        if frame.get_pixel_format() == opencv_display_format:
                                display = frame
                        else:

                                display = frame.convert_pixel_format(opencv_display_format)
                        self.display_queue.put(display.as_opencv_image(), True)

                cam.queue_frame(frame)
```

triggerCamera and trigger Functions:

- triggerCamera: Configures the camera for software triggering.
- trigger: Sends a software trigger signal to the camera after a short delay (0.05 seconds).

```
def triggerCamera(camera):
        camera.TriggerSourceSet('Software')
        camera.TriggerModeSet('On')
        camera.TriggerSelector.set('FrameStart')
        camera.AcquisitionMode.set('continious')

def trigger(camera):
        time.sleep(0.05)
        camera.TriggerSoftware.run()
```

Vimba System Context Manager:

- with VmbSystem.get_instance() as vmb:: This line enters a context manager block using VmbSystem.get_instance(), which establishes a connection to the Vimba system. This is the entry point for interacting with Vimba-compatible cameras. The variable vmb is used to reference the Vimba system object within the block.

```
with VmbSystem.get_instance() as vmb:
        ## get access to the cameras
```

Camera Access and Setup:

- `with get_camera()[0] as camL, get_camera()[1] as camR:` This nested context manager block uses the `get_camera()` function (defined earlier) to retrieve the first two detected cameras and assigns them to the variables `camL` (left camera) and `camR` (right camera). The `with` statement ensures that the cameras are properly opened and closed.
- `setup_camera(camR)`, `setup_camera(camL)`: Calls the `setup_camera` function for each camera to configure their settings (exposure, white balance, etc.).

```
img= None,
with get_camera()[0] as camL, get_camera()[1] as camR:

        ## setup cameras parameters
        setup_camera(camR)
        setup_camera(camL)
        triggerCamera(camL)
        triggerCamera(camR)
```

Image Handler Initialization:

- `handlerR = Handler()`, `handlerL = Handler()`: Creates instances of the custom `Handler` class for each camera. These handlers will be responsible for receiving image frames from the cameras and managing their processing and display.
- `blur = cv.GaussianBlur(img,(5,5),0)`: This applies a Gaussian blur to the image, for smoothening of the image.

```
handlerR = Handler()
handlerL = Handler()
blur = cv.GaussianBlur(img,(5,5),0) ## Gaussian blur for smoothness of the image
# manually  setting off camerra exposure  and  Time set
camR.ExposureAuto.set('Off')
camR.ExposureTime.set('20000')
camL.ExposureAuto.set('Off')
camL.ExposureTime.set('20000')

try:
        camR.UseSetSelector.set('Default')
        camL.UseSetSelector.set('Default')
except(AttributeError, VmbFeatueError):
        print('Failed to load')
        try:

                camR.UseSetLoad.run()
                camL.UseSetLoad.run()
        except(AttributeError, VmbFeatueError):
                print('Failed to run the Features')
```

Video Streaming and Processing:

- try...finally...: A try...finally block is used to ensure that the cameras are always stopped gracefully, even if an error occurs during streaming.
- camR.start_streaming(handler=handlerR, buffer_count=10) and camL.start_streaming(handler=handlerL, buffer_count=10): These lines initiate the continuous image streaming from both cameras. The handler argument specifies the Handler object that will receive the image frames. The buffer_count sets the number of frames that can be buffered before the camera stops acquiring new ones (to prevent memory issues).
- The # (Main image acquisition, processing, and display loop) comment indicates where the core image processing and display logic would be placed. This could involve:
    - Retrieving images from the Handler objects.
    - Converting images to grayscale.
    - Undistorting and rectifying images (if calibration data is available).
    - Calculating the disparity map using SGBM or other algorithms.
    - Displaying the images and/or disparity map in windows.

```
## video start streaming
try:
        camR.start_streaming(handler = handlerR, buffer_count = 10)
        camL.start_streaming(handler = handlerL, buffer_count = 10)

        msg = 'Stream from \'{}\. Press Esc key to stop stream'
        escKey = 27
        counter = 0
        size = (640,480)

        while True:

                ## get the image
                displayR = handlerR.get_image()
                displayL = handlerL.get_image()

                ##resize the image to 640x480
                imageR = cv.resize(displayR, size)
                imageL = cv.resize(displayL, size)

                grayR = cv.cvtColor(imageR, cv.COLOR_BGR2GRAY)
                grayL = cv.cvtColor(imageL, cv.COLOR_BGR2GRAY)
                cv.imshow('Right Camera'+ msg.format(camL.get_name()), grayR)
                cv.imshow('Left Camera' + msg.format(camL.get_name()), grayL)


                key = cv.waitKey(1)
                if key == escKey:
                          cv.destroyWindow(msg.format(camR.get_name()))
                          cv.destroyWindow(msg.format(camL.get_name()))
                        camR.stop_streaming()
                        camL.stop_streaming()
                        cv.destroyAllWindows()
                        break

                elif key == ord('s'):
                        print('Images Saved!!!')
                        counter= counter+1
                        cv.imwrite('S:\Bin Picking Project\TEST\ImgL_'+str(counter)+'.bmp',grayL)
                        cv.imwrite('S:\Bin Picking Project\TEST\ImgR_'+str(counter)+'.bmp',grayR)



finally:
        camR.stop_streaming()
        camL.stop_streaming()
```

Stereo Calibration – The Big Picture

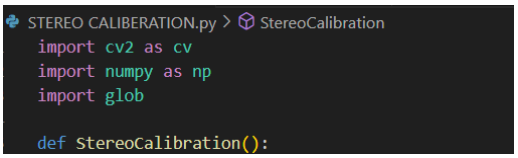
The goal of stereo calibration is to determine the precise relationship between the two cameras in the stereo vision setup. This relationship is crucial for accurately converting disparity maps (which represent the difference in pixel positions of the same object in the left and right images) into meaningful depth information.

Working:

1. Individual Camera Calibration (Intrinsic Parameters): Before calibrating the stereo system, you calibrate each camera individually. This involves finding the internal parameters of the camera, such as:
    - Focal Length: How strongly the lens converges or diverges light.
    - Principal Point: The center point where the optical axis intersects the image plane.
    - Distortion Coefficients: Parameters that model lens distortions (like barrel or pincushion distortion).
2. Stereo Calibration (Extrinsic Parameters): After calibrating the individual cameras, the stereo calibration process focuses on finding the extrinsic parameters of the stereo system:
    - Rotation Matrix: Describes how the right camera is rotated relative to the left camera.
    - Translation Vector: Indicates the distance between the two cameras along each axis (X, Y, and Z).

Code Explanation:

- A function named `StereoCalibration` that encapsulates the entire calibration process.This helps organize the code and makes it reusable if needed.



This section sets up the parameters for chessboard corner detection:

- `boardsize`: Defines the number of internal corners on your calibration board.
- `framesize`: The resolution of your camera images.
- `criteria`: Sets the conditions for stopping the corner refinement process (maximum iterations or desired accuracy).
- `objp`: Calculates the 3D coordinates of the chessboard corners in the real world, assuming the board is on the Z=0 plane.
- The lists `objPoints`, `imgPointsL`, and `imgPointsR` will be used to store the detected corners across multiple images for both cameras.

```
######### FIND CHESSBOARD CORNERS ######################
boardsize = (9,7)
framesize = (640,480)

# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

#prepare object points with numpy
objP = np.zeros((boardsize[0]*boardsize[1],3), np.float32)
objP[:,:2] = np.mgrid[0:boardsize[0],0:boardsize[1]].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objPoints = [] # 3d point in real world space
imgPointsL = [] # 2d points in image plane of camera1.
imgPointsR = [] # 2d points in image plane of camera0.
```

Image Loading and Corner Finding:

- `glob.glob`: Finds all BMP image files in the specified directories for the left (CamL) and right (CamR) cameras.
- The `for` loop iterates through pairs of left and right images, loading them and finding the chessboard corners using `cv.findChessboardCorners`.
- If corners are found (`retL` and `retR` are `True`), the object points and refined image points are added to the respective lists.
- The corners are then drawn on the images and displayed for visual confirmation.

```
for imageL, imageR in zip(imagesL, imagesR):

    imgL = cv.imread(imageL)
    imgR = cv.imread(imageR)

    grayL = cv.cvtColor(imgL, cv.COLOR_BGR2GRAY)
    grayR = cv.cvtColor(imgR, cv.COLOR_BGR2GRAY)


    #Find corners in the chessboard
    retL, cornersL = cv.findChessboardCorners(grayL, boardsize, None)
    retR, cornersR = cv.findChessboardCorners(grayR, boardsize, None)

    print(retL, retR)
    # If corners are found, add object points and image points
    if retL and retR == True:
        objPoints.append(objP)

        #append the chessboard corner point to imagepoints
        cornersL = cv.cornerSubPix(grayL, cornersL, (11,11),(-1,-1), criteria)
        imgPointsL.append(cornersL)
        cornersR = cv.cornerSubPix(grayR, cornersR, (11,11),(-1,-1), criteria)
        imgPointsR.append(cornersR)


        # draw and display the corners
        cv.drawChessboardCorners(imgL, boardsize,cornersL, retL)
        cv.imshow("LEFT_img", imgL)
        cv.drawChessboardCorners(imgR, boardsize,cornersR, retR)
        cv.imshow("RIGHT_img", imgR)


        cv.waitKey(3000)

cv.destroyAllWindows()
```

Camera Calibration:

- `cv.calibrateCamera`: Calibrates each camera individually using the collected object points and image points. This estimates the intrinsic parameters (camera matrix and distortion coefficients) for each camera.
- `cv.getOptimalNewCameraMatrix`: Calculates an improved camera matrix and a region of interest (ROI) to remove some distortion from the images.

```
#############   CALIBRATION ################################################
#Camera1 Caliberation
calibration1L, cameraMatrixL, distortionL, rotVectorL, transVectorL = cv.calibrateCamera(objPoints, imgPointsL, grayL.shape[::-1], None, None)
heightL, widthL, channelsL = imgL.shape
#  # #### Returns the new camera intrinsic matrix based on the free scaling parameter.
newCameraMatrixL, roi_L = cv.getOptimalNewCameraMatrix(cameraMatrixL, distortionL,(widthL, heightL), 1, (widthL, heightL))

# Camera2 Caliberation
calibrationR, cameraMatrixR, distortionR, rotVectorR, transVectorR = cv.calibrateCamera(objPoints, imgPointsR, grayR.shape[::-1], None, None)
heightR, widthR, channelsR = imgR.shape
#  # #### Returns the new camera intrinsic matrix based on the free scaling parameter.
newCameraMatrixR, roi_R = cv.getOptimalNewCameraMatrix(cameraMatrixR, distortionR,(widthR, heightR), 1, (widthR, heightR))
```

Stereo Calibration:

- `cv.stereoCalibrate`: Calibrates the stereo camera system. It determines the relative rotation (`rotMatrix`) and translation (`transVector`) between the two cameras. This information is essential for stereo rectification and depth estimation.

```
################ Stereo Vision Caliberation ######################
flags = 0
flags |= cv.CALIB_FIX_INTRINSIC
# here we fix the intrinsic camera matrices so that only Rotation, tanslataion essential and fundamental matrices
# so the intrinsic parameters are the same

ret, newCameraMatrixL, distCoeffsL, newCameraMatrixR, distCoeffsR, rotMatrix, transVector, essMatrix, fundMatrix = cv.stereoCalibrate(objPoints,
                                                                                                                                        newCameraMatrixR, distort
```

Stereo Rectification:

- `cv.stereoRectify`: Computes the rectification transformations (`rectTransL`, `rectTransR`) that make the image planes of both cameras parallel. This is necessary for accurate disparity calculation.
- `cv.initUndistortRectifyMap`: Creates maps for remapping pixels in the original images to the rectified space.

```
############ Stereo Rectification #####################################
# #  Computes rectification transforms for each head of a calibrated stereo camera
# # # alpha = 1
# # #alpha=0 means that the rectified images are zoomed and shifted
rectifyScale = 1
rectTransL, rectTransR, projMatrixL, projMatrixR, disp2DepthMapMatrix, roiL, roiR = cv.stereoRectify(newCameraMatrixL, distCoeffsL, newCameraMat


stereoMapLx, stereoMapLy = cv.initUndistortRectifyMap(cameraMatrixL, distCoeffsL, rotMatrix, projMatrixL, grayL.shape[::-1], cv.CV_32FC1)

stereoMapRx, stereoMapRy = cv.initUndistortRectifyMap(cameraMatrixR, distCoeffsR, rotMatrix, projMatrixR, grayR.shape[::-1], cv.CV_32FC1)
```

Saving Calibration Parameters:

- Calibration results are saved to `.npy` files for individual parameters and a `.xml` file for the stereo maps. This allows you to reuse these parameters without recalibrating every time.

```
    print("Saving Caliberation Parameters")

    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\cameraMatrixLeft.npy', cameraMatrixL)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\cameraMatrixRight.npy', cameraMatrixR)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\newcameraMatrixLeft.npy', newCameraMatrixL)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\newcameraMatrixRight.npy', newCameraMatrixR)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\Distortion_CoefficentLeft.npy', distCoeffsL)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\Distortion_CoefficentRight.npy', distCoeffsR)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\RotationMatrixLR.npy', rotMatrix)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\TranslationVectorLR.npy', transVector)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\projectionMatrixLeft.npy', projMatrixL)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\projectionMatrixRight.npy', projMatrixR)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\rotationMatrixLeft.npy', rectT
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\rotationMatrixRight.npy', rect
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\disparity2depthMapMatrix.npy',disp2DepthMapMatrix)



    print("saving Parameters")
    cv_file = cv.FileStorage('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\stereoMap.xml', cv.FILE_STORAGE_WRITE)

    cv_file.write('stereoMapL_x', stereoMapLx)
    cv_file.write('stereoMapL_y', stereoMapLy)
    cv_file.write('stereoMapR_x', stereoMapRx)
    cv_file.write('stereoMapR_y', stereoMapRy)

    cv_file.release()



stereoCalibration()
```

(variable) disp2DepthMapMatrix: Any

OBSERVATIONS:

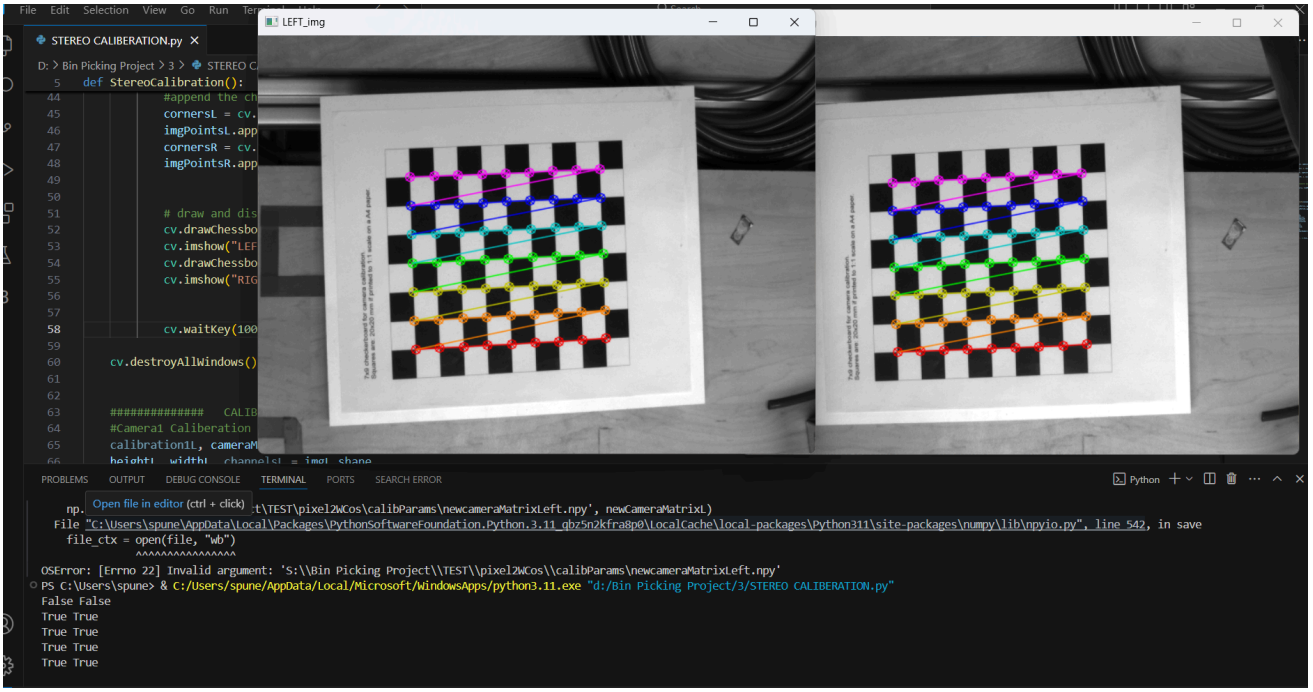


Fig. 1: Chessboard Corners Detection

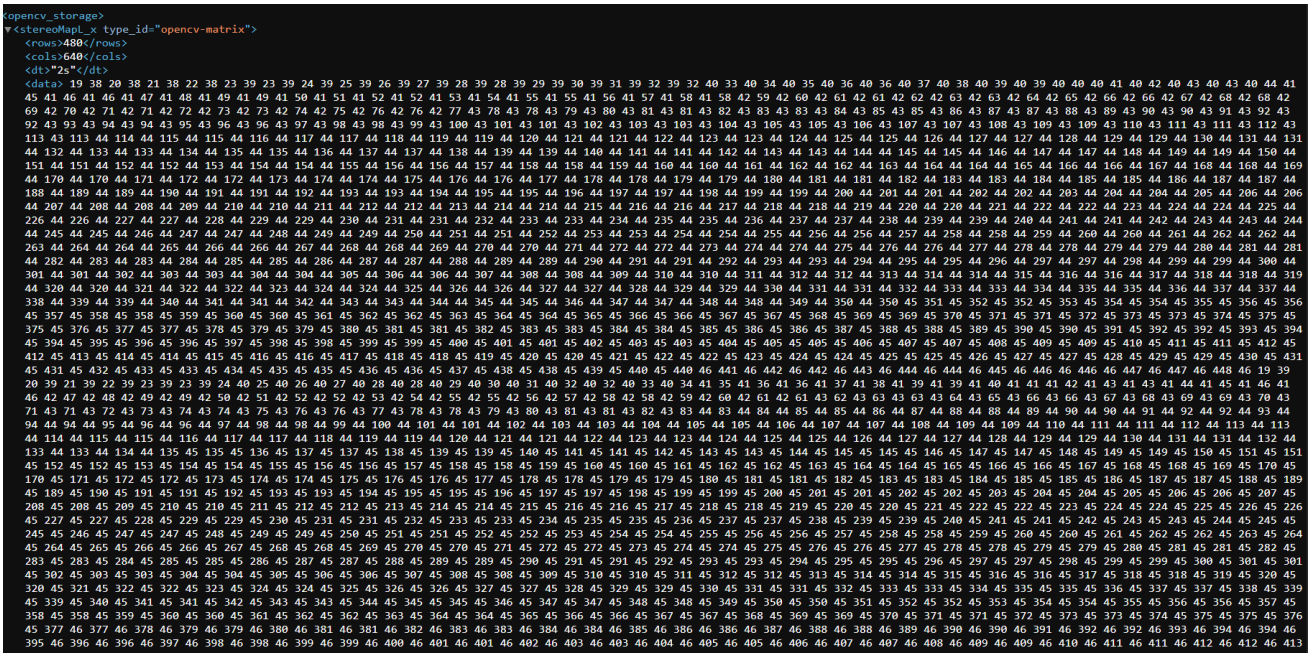


Fig. 2: Calibration Parameter Matrix

Stereo Rectification

This code is designed to load stereo calibration parameters (calculated earlier calibration script) and use them to perform two key operations:

1. **Undistortion:** Correcting lens distortion present in the raw images from the left and right cameras.
2. **Rectification:** Transforming the undistorted images so that corresponding points lie on the same horizontal line (as explained earlier).

Breakdown of Functions:

- **`undistort(image0, image1)`:**
    - **Input:** Takes the raw (distorted) images from the left (image0) and right (image1) cameras.
    - **Process:** Loads camera matrices (`camMatrix0`, `camMatrix1`) and distortion coefficients (`distcoef0`, `distcoef1`) from the saved calibration files.

      Also loads the optimal new camera matrices (`newCamMat0`, `newCamMat1`), which might be slightly adjusted from the original ones to minimize unwanted pixels after undistortion.

      Applies OpenCV's `cv.undistort` function to each image, using the corresponding camera matrix and distortion coefficients.
    - **Output:** Returns the undistorted versions of the left and right images.

- **`undistortRectify(imageL, imageR)`:**
  Input: Takes the undistorted images from the left (imageL) and right (imageR) cameras.
  Process: Loads additional calibration parameters:
    - Rectification transformation matrices (`rectTransL`, `rectTransR`)
    - Projection matrices (`projMatrixL`, `projMatrixR`)

  Active : Re-computes the stereo maps using `cv.initUndistortRectifyMap` for each camera. This step combines undistortion and rectification into a single transformation.

  Applies the stereo maps using `cv.remap` to rectify (and undistort, if not done before) both images.

Output: Returns the rectified (and undistorted) versions of the left and right images, where corresponding points are now aligned on the same row.

Workflow:

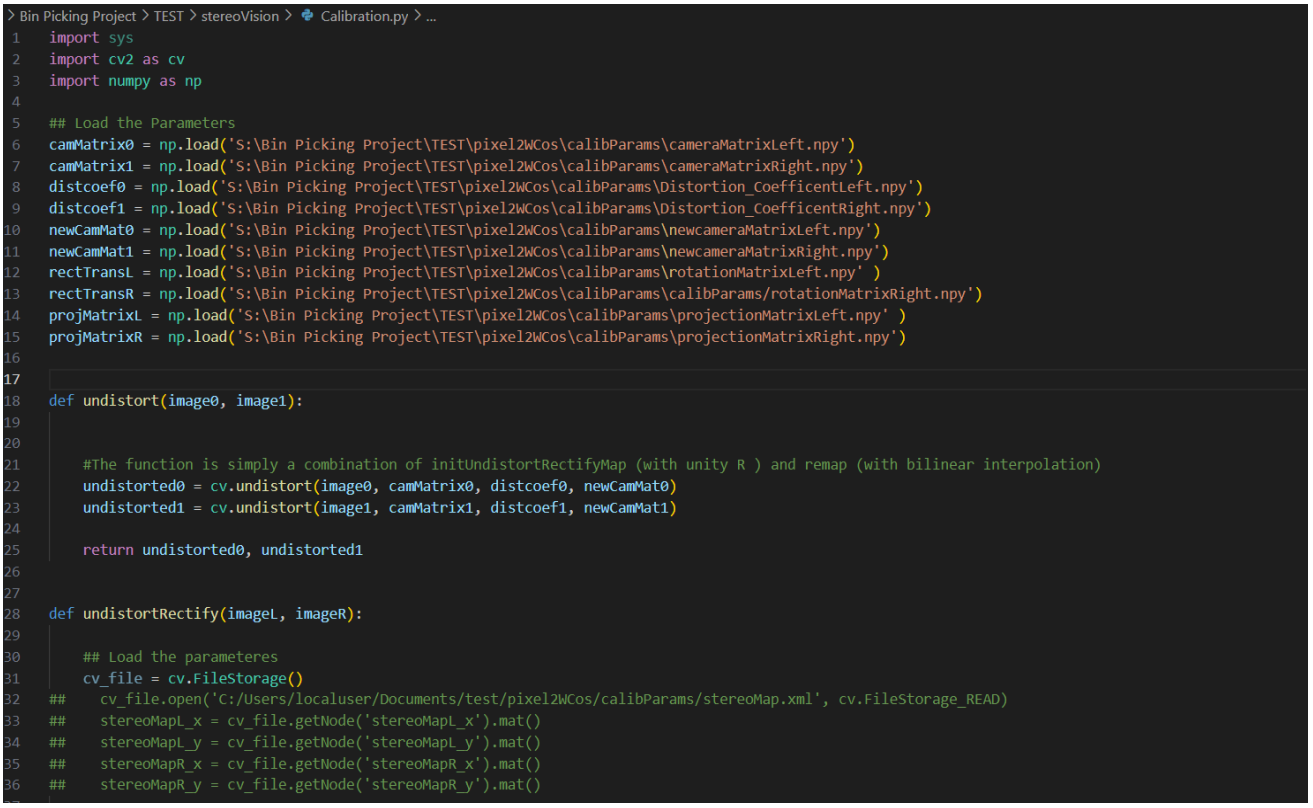
1. **Stereo Calibration:** You first run the calibration script you provided earlier to calculate and save all the necessary parameters.
2. **Load Parameters:** This new code snippet loads those saved parameters.
3. **Undistort and Rectify:** You then use these functions on pairs of stereo images:
    - Call `undistort` on the raw images.

- Call `undistortRectify` on the undistorted images (or directly on the raw images if you're recomputing the maps).

4. **Stereo Matching:** Now you have rectified images that are ready for stereo matching algorithms (like block matching or semi-global block matching) to compute disparity maps and ultimately depth information.

Key Points:

- **Flexibility:** The code provides two ways to handle rectification: using pre-computed maps for speed or recomputing them on-the-fly for flexibility.
- **Dependency:** This code relies entirely on the calibration parameters obtained from the first script.
- **Purpose:** This step is essential for most stereo vision applications, as rectified images greatly simplify the task of finding corresponding points, leading to more accurate depth estimation.



```
> Bin Picking Project > TEST > stereoVision > Calibration.py > ...
import sys
import cv2 as cv
import numpy as np

## Load the Parameters
camMatrix0 = np.load('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\cameraMatrixLeft.npy')
camMatrix1 = np.load('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\cameraMatrixRight.npy')
distcoef0 = np.load('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\Distortion_CoefficentLeft.npy')
distcoef1 = np.load('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\Distortion_CoefficentRight.npy')
newCamMat0 = np.load('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\newcameraMatrixLeft.npy')
newCamMat1 = np.load('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\newcameraMatrixRight.npy')
rectTransL = np.load('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\rotationMatrixLeft.npy' )
rectTransR = np.load('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\calibParams/rotationMatrixRight.npy')
projMatrixL = np.load('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\projectionMatrixLeft.npy' )
projMatrixR = np.load('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\projectionMatrixRight.npy')


def undistort(image0, image1):


    #The function is simply a combination of initUndistortRectifyMap (with unity R ) and remap (with bilinear interpolation)
    undistorted0 = cv.undistort(image0, camMatrix0, distcoef0, newCamMat0)
    undistorted1 = cv.undistort(image1, camMatrix1, distcoef1, newCamMat1)

    return undistorted0, undistorted1


def undistortRectify(imageL, imageR):

    ## Load the parameteres
    cv_file = cv.FileStorage()
##    cv_file.open('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/stereoMap.xml', cv.FileStorage_READ)
##    stereoMapL_x = cv_file.getNode('stereoMapL_x').mat()
##    stereoMapL_y = cv_file.getNode('stereoMapL_y').mat()
##    stereoMapR_x = cv_file.getNode('stereoMapR_x').mat()
##    stereoMapR_y = cv_file.getNode('stereoMapR_y').mat()
```

Disparity:

Disparity is a fundamental concept in stereo vision, referring to the difference in image coordinates of a point between the left and right camera views. This difference arises due to the horizontal separation (baseline) between the two cameras. By analyzing the disparity values in a pair of rectified stereo images, we can compute a disparity map, which is a representation of the scene's depth information. Larger disparities indicate closer objects, while smaller disparities correspond to objects farther away from the camera. Disparity estimation is crucial for tasks like 3D reconstruction, depth sensing, and obstacle detection in applications such as autonomous navigation, robotics, and augmented reality.

Code Explanation:

Import Libraries: Import the necessary libraries for image processing (OpenCV), numerical calculations (NumPy), file management (glob, sys), and visualization (Matplotlib).

```
import cv2 as cv
import numpy as np
import glob
import sys
import matplotlib.pyplot as plt
```

Calibration Parameters:

- `boardSize`: Define the dimensions of the checkerboard pattern used for calibration. This should match your physical checkerboard.
- `frameSize`: Set the resolution of the camera images.
- `criteria`: Termination criteria for the corner refinement algorithm (`cv2.cornerSubPix`). The algorithm stops iterating when either the specified accuracy (`0.001`) is reached or the maximum number of iterations (`30`) is exceeded.

```
# Calibration Parameters (Modify as per your setup)
boardSize = (7, 9)  # Number of inner corners on the chessboard (horizontal, vertical)
frameSize = (640, 480)  # Resolution (width, height) of the camera images
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)  # Stopping criteria for calibration accuracy
```

Object Points (3D):

- `objp`: Creates a 3D array to store the coordinates of the checkerboard corners in real-world space. Since we know the dimensions of the checkerboard and assume it's flat, we set the Z-coordinate of all corners to 0.
- `np.mgrid`: Generates a 2D grid of coordinates for the checkerboard corners in the X-Y plane.
- `T.reshape`: Reshapes the coordinates into a format suitable for the calibration function.
- Initialize empty lists to store the object points (3D) and image points (2D) detected in the calibration images.

```
# Prepare 3D Object Points (Chessboard Corners in Real-World Coordinates)
objp = np.zeros((boardSize[0] * boardSize[1], 3), np.float32)  # Create an array for object points
objp[:, :2] = np.mgrid[0:boardSize[0], 0:boardSize[1]].T.reshape(-1, 2)  # Fill the array with corner coordinates

# Arrays to Store Points from Calibration Images
objPoints = []  # 3D points in real world space
imgPointsL = []  # 2D points in left image plane
imgPointsR = []  # 2D points in right image plane
```

`calibrate_camera` Function:

- This function takes a list of calibration image paths, the checkerboard size, frame size, and termination criteria as input.
- For each image:
  - Loads the image using `cv2.imread`.
  - Converts it to grayscale using `cv2.cvtColor`.
  - Finds the checkerboard corners using `cv2.findChessboardCorners`.
  - If corners are found, the object points (`objp`) and the refined corner locations (`corners2`) are added to their respective lists.
  - The checkerboard corners are drawn on the image using `cv2.drawChessboardCorners` for visual confirmation.
- Finally, it uses `cv2.calibrateCamera` to calculate the intrinsic camera parameters (camera matrix and distortion coefficients) based on the collected object and image points.

```
# --- Function to Calibrate the Camera ---
def calibrate_camera(images, boardSize, frameSize, criteria):
    objpoints = []
    imgpoints = []

    for image in images:
        img = cv.imread(image)
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

        # Find the chess board corners
        ret, corners = cv.findChessboardCorners(gray, boardSize, None)

        # If found, add object points, image points (after refining them)
        if ret:
            objpoints.append(objp)

            corners2 = cv.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)
            imgpoints.append(corners2)

            # Draw and display the corners
            img = cv.drawChessboardCorners(img, boardSize, corners2, ret)
            cv.imshow('img', img)
            cv.waitKey(500)

    cv.destroyAllWindows()
    ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, frameSize, None, None)
    return ret, mtx, dist, rvecs, tvecs
```

`depth_map` Function:

- This function calculates the disparity and depth map from rectified stereo images (`imgL` and `imgR`).
- It takes the stereo rectification maps (`leftMapX`, `leftMapY, rightMapX, rightMApY`) and the disparity-to-depth mapping matrix (`Q`) as input from the calibration.
- It uses the Semi-Global Block Matching (SGBM) algorithm (`cv2.StereoSGBM_create`) to compute the disparity map as it's a popular algorithm for calculating disparity maps in stereo vision. It strikes a balance between accuracy and computational efficiency, making it suitable for real-time applications. working of SGBM algorithm:

1. **Matching Cost Computation:**
    - The algorithm divides the input images into small blocks (e.g., 3x3 pixels in this code).
    - For each block in the left image, it compares it to candidate blocks in the right image, calculating a matching cost for each potential correspondence.
    - The matching cost measures how similar the two blocks are (e.g., using the Sum of Absolute Differences – SAD).

2. **Cost Aggregation:**
    - SGBM aggregates the matching costs along multiple paths (typically 8 or 16 directions) for each pixel. This "semi-global" aggregation helps to improve the accuracy and robustness of the disparity estimates, reducing noise and smoothing out discontinuities.
    - The aggregation step considers both the local matching costs and smoothness constraints to find the optimal disparity for each pixel.

3. **Disparity Optimization:**
    - After aggregation, the algorithm finds the disparity value with the lowest aggregated cost for each pixel. This is the final disparity estimate, representing the horizontal shift between the corresponding points in the left and right images.
    - Additional post-processing steps, such as subpixel interpolation and outlier removal, can be applied to further refine the disparity map.

SGBM Parameters in the Code

1. `minDisparity`: The minimum possible disparity value (negative values allow for objects closer to the camera than the plane of focus).
2. `numDisparities`: The number of disparity levels to consider. A higher value increases the maximum measurable depth but also increases computational complexity.
3. `blockSize`: The size of the blocks used for matching. Larger blocks may be more robust to noise but may also lose finer details.
4. `P1`, `P2`: Penalty parameters for disparity changes between neighboring pixels. They control the smoothness of the disparity map.

5. **`disp12MaxDiff`:** This parameter controls the maximum allowed difference in disparity values between corresponding pixels in the left and right images that are 12 pixels apart in the scanline. Setting it to a higher value allows for more flexibility in handling areas with repetitive patterns or textureless regions, but it may also introduce more noise in the disparity map. A value of -1 disables this check, which is the default in OpenCV.
6. **`uniquenessRatio`:** This parameter influences the uniqueness of the selected matches during cost aggregation. A higher value enforces stricter uniqueness, meaning that a pixel in the left image is less likely to be matched to multiple pixels in the right image. This can reduce the number of false matches, especially in low-texture areas, but it might also lead to some valid matches being rejected.
7. **`speckleWindowSize`:** This parameter defines the size of a region used for speckle filtering. Speckle filtering aims to remove small, isolated regions in the disparity map that are likely caused by noise or mismatches. A larger window size removes larger speckles, but it might also smooth out legitimate details in the scene.
8. **`speckleRange`:** This parameter sets the maximum allowed disparity variation within a speckle region. If the disparity variation within a region exceeds this value, the region is considered a speckle and is removed. A higher value allows for more disparity variation within a region, potentially preserving more details, but it also increases the risk of preserving noise.
9. **`preFilterCap`:** This parameter limits the range of intensity values in the pre-filtering step of SGBM. It helps to reduce noise and improve the accuracy of the matching cost computation, especially in images with high dynamic range.
10. **`mode`:** This parameter selects the SGBM mode, determining the specific algorithm variant to be used.
    - `cv.STEREO_SGBM_MODE_SGBM`: The standard SGBM mode.
    - `cv.STEREO_SGBM_MODE_HH`: A mode that emphasizes horizontal smoothness (used for images with significant horizontal structures).
    - `cv.STEREO_SGBM_MODE_SGBM_3WAY`: This mode computes three disparity maps (using different penalty parameters) and averages them, potentially improving accuracy in some cases (this is the mode used in your code).

```python
# --- Function to Calculate Depth Map from Stereo Images ---
def depth_map(imgL, imgR, Q, leftMapX, leftMapY, rightMapX, rightMapY):
    window_size = 3  # SGBM Parameters

    left_matcher = cv.StereoSGBM_create(
        minDisparity=-1,
        numDisparities=5*16,
        blockSize=window_size,
        P1=8 * 3 * window_size,
        P2=32 * 3 * window_size,
        disp12MaxDiff=12,
        uniquenessRatio=10,
        speckleWindowSize=50,
        speckleRange=32,
        preFilterCap=63,
        mode=cv.STEREO_SGBM_MODE_SGBM_3WAY
    )
```

`right_matcher = cv.ximgproc.createRightMatcher(left_matcher)`

- **Purpose:** In stereo vision, the left and right matchers are used to compute disparity maps for both images independently. While the `left_matcher` calculates disparity values for the left image (how much each pixel in the left image is shifted to the right compared to its corresponding pixel in the right image), the `right_matcher` does the opposite – it calculates the disparity for the right image (how much each pixel is shifted to the left compared to its corresponding pixel in the left image).
- **`cv.ximgproc.createRightMatcher(left_matcher)`:** This OpenCV function creates a right matcher that is specifically designed to be compatible with the given `left_matcher`. The matching costs and parameters of the right matcher are automatically configured to mirror those of the left matcher, ensuring consistency in the disparity calculations.

Need of two matchers:-

Using two matchers provides a form of cross-checking to improve the quality of the disparity map. The disparity values obtained from both matchers can be compared and combined to identify and correct errors. For example, if the left matcher finds a disparity of 5 for a pixel, but the right matcher finds a disparity of -3 for the same pixel, it's likely that one of these values is incorrect. The WLS filter, which is applied later, leverages the information from both matchers to refine the disparity estimates.

`lmbda = 80000` (Lambda)

- **Purpose:** This parameter (lambda) is used by the Weighted Least Squares (WLS) filter. The WLS filter aims to smooth out the disparity map while preserving object boundaries. Lambda controls the relative weight given to the data term (the raw disparity values) versus the smoothness term (how much neighboring disparities should influence each other).
- **Effect of High Lambda:** A higher lambda value like 80000 emphasizes smoothness, leading to a smoother disparity map but potentially over-smoothing some fine details. This can be beneficial for reducing noise, especially in low-texture regions.

`sigma = 1.3` (Sigma)

- **Purpose:** This parameter (sigma) is also used by the WLS filter. It controls the sensitivity of the filter to color differences between pixels. A larger sigma value means that the filter is less sensitive to color differences, leading to more aggressive smoothing.
- **Effect of Low Sigma:** A smaller sigma value like 1.3 makes the filter more sensitive to color changes, allowing it to preserve edges and object boundaries more effectively during the smoothing process.

```
right_matcher = cv.ximgproc.createRightMatcher(left_matcher)
lmbda = 80000
sigma = 1.3
```

Image Rectification:

1. **Purpose:** The `cv2.remap` function is used to rectify the left (`imgL`) and right (`imgR`) stereo images. Rectification is a geometric transformation that warps the images so that corresponding points (points representing the same 3D object in the scene) appear on the same horizontal line in both images. This simplifies the subsequent disparity calculation step.

   Inputs:

   - `imgL`, `imgR`: The original left and right stereo images.
   - `leftMapX`, `leftMapY`, `rightMapX`, `rightMapY`: These are mapping matrices obtained from the stereo rectification process (`cv2.stereoRectify`). They define the pixel transformations required to warp the images.

   Output:

   - `imgL`, `imgR`: The rectified left and right images.

   Interpolation: `cv2.INTER_LINEAR` indicates that linear interpolation should be used to fill in the values of pixels that fall between integer coordinates after the remapping process.

```
# Rectify Images
imgL = cv.remap(imgL, leftMapX, leftMapY, cv.INTER_LINEAR)
imgR = cv.remap(imgR, rightMapX, rightMapY, cv.INTER_LINEAR)
```

Disparity Calculation:

1. **Purpose:** This step computes the disparity map, which encodes the horizontal displacement between corresponding points in the rectified left and right images. Disparity is directly related to depth—larger disparities correspond to closer objects.

`left_matcher.compute(imgL, imgR)`:

- `left_matcher`: This is an instance of the Semi-Global Block Matching (SGBM) algorithm (`cv2.StereoSGBM_create`) that has been configured earlier in the code.
- This line performs the actual disparity calculation using the SGBM algorithm on the rectified images.

Output:

- Disparity map: 16-bit image showing horizontal shift of each pixel.

```
# Disparity Calculation and Filtering
disparity = left_matcher.compute(imgL, imgR)
disparity = np.int16(disparity)
filteredImg = wls_filter.filter(disparity, imgL, None, None)
```

- Code applies a Weighted Least Squares (WLS) filter
  `cv2.ximgproc.createDisparityWLSFilter()`: Creates a WLS filter object.
  `wls_filter.filter(disparity, imgL, None, None)`: Applies the WLS filter to the raw disparity map, using the left image as a reference to preserve object boundaries. This filtering step further improves the accuracy of the disparity estimates.
- Finally, it uses the `Q` matrix to reproject the disparity map into 3D points (`cv2.reprojectImageTo3D`) to obtain a depth map then the disparity map is normalized and converted to a uint8 format for visualization.

```
# Convert to Depth Map
points_3D = cv.reprojectImageTo3D(filteredImg, Q)
```

Main Program:

- Check Arguments: The code first ensures that the user has provided the correct command-line arguments when running the script. It expects two arguments: the file path for the left image and the file path for the right image. If the arguments are missing, it prints a usage message and exits.

```
# --- Main Program ---
if __name__ == "__main__":
    # Check if image paths were provided as arguments
    if len(sys.argv) < 3:
        print("Usage: python your_script.py <left_image_path> <right_image_path>")
        sys.exit(1)

    left_image_path = sys.argv[1]
    right_image_path = sys.argv[2]
```

Load Stereo Images:

- It loads the right and left images using `cv2.imread`. The `imageSize` variable stores the dimensions (height and width) of the left image, which should be the same as the right image since they are a stereo pair.

```
# Load and Preprocess Images (After Calibration)
imagesR = cv.imread(right_image_path)
imagesL = cv.imread(left_image_path)

if imagesR is None or imagesL is None:
    raise ValueError("Error loading images. Check file paths and ensure images are valid.")

imageSize = imagesL.shape[:2]
```

Load Calibration Images:

- It uses `glob.glob` to find all the calibration images for both the left (`CamL`) and right (`CamR`) cameras, which are saved as `.bmp` files in the specified directories.

```
# Load Images (For Both Cameras)
imagesL_cali = glob.glob(r"D:\Bin Picking Project\3\Cam1\*.bmp")
imagesR_cali = glob.glob(r"D:\Bin Picking Project\3\Cam0\*.bmp")
```

Camera Calibration:

- It calls the `calibrate_camera` function twice (once for each camera) to determine the intrinsic camera parameters (camera matrices and distortion coefficients) using the loaded calibration images.

```
# Camera Calibration
retL, mtxL, distL, rvecsL, tvecsL = calibrate_camera(imagesL_cali, boardSize, frameSize, criteria)
retR, mtxR, distR, rvecsR, tvecsR = calibrate_camera(imagesR_cali, boardSize, frameSize, criteria)
```

Calibration Check:

- Verifies that at least two valid calibration images were found for each camera. If not, it displays an error message and exits.

```
# Check if there are enough points for calibration
if len(objPoints) < 2:
    print("Not enough calibration images found. Please use at least 2 images.")
    exit()
```

Stereo Calibration:

- Performs stereo calibration using `cv2.stereoCalibrate` with fixed intrinsic parameters to find the relative rotation and translation between the cameras.
- `cv2.stereoCalibrate` is used to find the relative rotation (`R`) and translation (`T`) between the two cameras, along with other parameters like the essential matrix (`E`) and fundamental matrix (`F`).
- It checks if the stereo calibration was successful, raising an error if it failed.

```python
# Stereo Calibration
flags = 0
flags |= cv.CALIB_FIX_INTRINSIC
criteria_stereo = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

retS, newCameraMatrixL, distCoeffsL, newCameraMatrixR, distCoeffsR, R, T, E, F = cv.stereoCalibrate(
    objPoints, imgPointsL, imgPointsR, mtxL, distL, mtxR, distR,
    imageSize, criteria_stereo, flags
)
if not retS:
    raise ValueError("Stereo calibration failed. Check your calibration images and settings.")
```

Stereo Rectification:

- `cv2.stereoRectify` computes rectification transforms to correct lens distortion and align the stereo images for disparity calculation.
- `cv2.initUndistortRectifyMap` generates mapping matrices (`left_map1`, `left_map2`, etc.) for remapping the images during rectification.

```python
# Stereo Rectification
rectify_scale = 1
R1, R2, P1, P2, Q, validPixROI1, validPixROI2 = cv.stereoRectify(
    newCameraMatrixL, distCoeffsL, newCameraMatrixR, distCoeffsR,
    imageSize, R, T, rectify_scale, (0, 0)
)
left_map1, left_map2 = cv.initUndistortRectifyMap(newCameraMatrixL, distCoeffsL, R1, P1, imageSize, cv.CV_16SC2)
right_map1, right_map2 = cv.initUndistortRectifyMap(newCameraMatrixR, distCoeffsR, R2, P2, imageSize, cv.CV_16SC2)
```

Disparity and Depth Calculation:

- Calls the `depth_map` function to compute the disparity map and the 3D points using the rectified images and the `Q` matrix.

```python
# Calculate Disparity and Depth Map (Pass rectification maps)
disparity, points_3D = depth_map(imagesL, imagesR, Q, left_map1, left_map2, right_map1, right_map2)
```

Display Disparity Map:

- Visualizes the calculated disparity map using Matplotlib. Brighter areas in the map represent objects closer to the camera, while darker areas represent objects farther away.

```python
# Visualization
plt.imshow(disparity, 'gray')
plt.title('Disparity Map')
plt.show()
```

Stereo Vision Depth Estimation: An Interactive Exploration of Disparity Maps

This graphical user interface (GUI) allows users to interactively explore the process of depth estimation in stereo vision. By adjusting parameters of the Semi-Global Block Matching (SGBM) algorithm, users can observe the resulting disparity map, a visual representation of depth information, in real-time. This tool provides a hands-on way to understand how different parameters affect the accuracy and quality of depth estimation, making it valuable for both educational and research purposes in fields such as robotics, autonomous navigation, and 3D reconstruction.

Key Features and Functionality

- **Intuitive Interface:** The GUI presents an intuitive interface with sliders for adjusting various parameters of the Semi-Global Block Matching (SGBM) algorithm, a widely used method for disparity estimation.
- **Real-Time Updates:** The disparity map is dynamically updated as the user adjusts parameters, providing immediate visual feedback on how different settings affect depth perception.
- **Color-Coded Visualization:** The disparity map is displayed using a vibrant JET colormap, where closer objects are represented in blue and farther objects in red, aiding in the interpretation of depth information.
- **Disparity Value Display:** Users can click on any point within the disparity map to view the precise disparity value at that location, aiding in the analysis and validation of results.
- **Parameter Sliders:** The GUI offers sliders for adjusting the following SGBM parameters:
    - `blockSize`: Size of the blocks used for matching.
    - `preFilterCap`: Intensity threshold for pre-filtering.
    - `minDisparity`: Minimum possible disparity value.
    - `numDisparities`: Number of disparity levels considered.
    - `preFilterSize`: Size of the pre-filtering kernel.
    - `textureThreshold`: Texture threshold for disparity computation.
    - `uniquenessRatio`: Enforces the uniqueness of matches.
    - `speckleRange`: Maximum disparity variation within a speckle.
    - `speckleWindowSize`: Size of the speckle filter window.
    - `P1`, `P2`: Penalty parameters for disparity change.
    - `disp12MaxDiff`: Maximum allowed difference in disparity for pixels 12 pixels apart.

Code Explanation:

1. Import Libraries:

- `tkinter` (aliased as `tk`): Provides the foundation for creating the graphical user interface (GUI) elements like windows, sliders, and canvases.
- `cv2` (OpenCV): Used for image loading, disparity calculation, and some visualization tasks.

- `numpy` (as `np`): Enables efficient numerical operations on the disparity map data.
- `PIL` (Python Imaging Library): Allows loading and manipulation of images in various formats.
- `ImageTk`: A module from PIL that bridges the gap between PIL images and Tkinter for display in GUI components.

```
import tkinter as tk
import cv2
import numpy as np
from PIL import Image, ImageTk
```

2. Load Stereo Images:
   - Attempts to load the left and right stereo images in grayscale format.
   - The `try-except` block handles potential errors during image loading, providing an error message and exiting the program if the images cannot be loaded.

```
# Load stereo images (BMP format)
try:
    left_image = cv2.imread("ImageL_1.bmp", cv2.IMREAD_GRAYSCALE)
    right_image = cv2.imread("ImageR_1.bmp", cv2.IMREAD_GRAYSCALE)
except Exception as e:
    print(f"Error loading images: {e}")
    exit()  # Exit the program if images can't be loaded
```

3. Check Image Loading:
   - Verifies that both images were successfully loaded. If either image is missing, an error message is printed and the program exits.

```
# Check if images were loaded successfully
if left_image is None or right_image is None:
    print("Error: Could not load one or both of the images.")
    exit()
```

4. Initialize Disparity Variables:
   - `disparity`: Will store the raw disparity map (16-bit signed integers) calculated by the SGBM algorithm.
   - `disparity_color`: Will store the colorized version of the disparity map for display.

```
# Initialize variables for disparity maps
disparity = None
disparity_color = None
```

5. Create SGBM Object:
   - Creates an instance of the OpenCV StereoSGBM class to handle disparity calculation using the SGBM algorithm.
   - `mode=3` specifies the use of the `STEREO_SGBM_MODE_SGBM_3WAY` mode, which computes three disparity maps with different parameters and averages them for potentially improved accuracy.

```
# Create StereoSGBM object for disparity calculation (SGBM_3WAY mode)
stereo = cv2.StereoSGBM_create(mode=3)
```

6. Mouse Callback Function:
   - `on_mouse`: This function is executed whenever the user clicks the left mouse button on the disparity map canvas.
   - It checks if the click is within the valid bounds of the disparity image.
   - It retrieves the disparity value at the clicked pixel, divides it by 16 to scale it to the correct units, and updates the `disparity_label` in the GUI to display the value.

```
# Mouse callback function to display disparity value at clicked point
def on_mouse(event, x, y, flags, param):
    global disparity
    if event == cv2.EVENT_LBUTTONDOWN and disparity is not None:
        # Check if click is within image bounds
        if 0 <= y < disparity.shape[0] and 0 <= x < disparity.shape[1]:
            # Disparity values are scaled by 16, so divide by 16 for actual value
            disparity_value = disparity[y, x] / 16.0
            if disparity_value > 0:  # Ensure valid disparity (positive)
                disparity_label.config(text=f"Disparity: {disparity_value:.2f}")
```

7. Update Disparity Function:
   - This function is called whenever any of the slider values change.
   - It reads the current values from all sliders and updates the parameters of the `stereo` (SGBM) object accordingly.
   - It recalculates the disparity map using the updated parameters (`stereo.compute`).
   - It normalizes the disparity map to a range of 0-255.
   - It applies the JET colormap to the normalized disparity map.
   - It converts the colored disparity map to a PIL `Image` and then to an `ImageTk.PhotoImage` for display in Tkinter.
   - It updates the disparity canvas to show the new disparity image.

```
# Set parameters for StereoSGBM algorithm
stereo.setBlockSize(block_size)
stereo.setPreFilterCap(preFilterCap_slider.get())
stereo.setMinDisparity(minDisparity_slider.get())
stereo.setNumDisparities(numDisparities_slider.get())
textureThreshold = textureThreshold_slider.get()
uniquenessRatio = uniquenessRatio_slider.get()
speckleRange = speckleRange_slider.get()
speckleWindowSize = speckleWindowSize_slider.get()
P1 = P1_slider.get()
P2 = P2_slider.get()
disp12MaxDiff = disp12MaxDiff_slider.get()

# Compute disparity map
disparity = stereo.compute(left_image, right_image)

# Normalize disparity for color mapping and display (0-255)
disparity_normalized = cv2.normalize(disparity, None, 0, 255, cv2.NORM_MINMAX, cv2.CV_8UC1)

# Apply JET colormap to the normalized disparity
disparity_color = cv2.applyColorMap(disparity_normalized, cv2.COLORMAP_JET)

# Convert disparity maps to PIL Image for display
disparity_image = Image.fromarray(disparity_color) # Use color map here
disparity_photo = ImageTk.PhotoImage(disparity_image)

# Update canvas with new disparity image
disparity_canvas.config(width=disparity_image.width, height=disparity_image.height)
disparity_canvas.delete("all")
disparity_canvas.create_image(0, 0, anchor=tk.NW, image=disparity_photo)
# Store reference to avoid garbage collection of the image
disparity_canvas.image = disparity_photo
disparity_canvas.bind("<Button-1>", on_mouse)  # Bind mouse events for value display
```

8. GUI Setup:
   - This section creates the main application window using `tk.Tk()` and sets its title to "Disparity Map GUI".
   - A frame (`slider_frame`) is created to hold all the parameter sliders. It is placed on the left side of the window using the grid layout manager.

9. Create Sliders:
   - A `create_slider` function is defined to simplify the creation of sliders. It takes the slider label, range, resolution, default value (optional), and length as input and returns a `tk.Scale` object.
   - Multiple sliders are created using this function for each SGBM parameter, and they are placed in the `slider_frame` one below the other.

```
def create_slider(label, from_, to, resolution=1, default=None, length=200):
    global slider_row  # Access the global slider_row variable

    slider = tk.Scale(slider_frame, from_=from_, to=to, resolution=resolution, orient=tk.HORIZONTAL,
                      label=label, length=length, command=update_disparity_map)
    if default is not None:
        slider.set(default)
    slider.grid(row=slider_row, column=0, sticky="ew", pady=5)

    slider_row += 1
    return slider

blockSize_slider = create_slider("blockSize (odd)", 3, 255, 2, 21)
preFilterCap_slider = create_slider("preFilterCap", 1, 63, default=29)
minDisparity_slider = create_slider("minDisparity", -100, 100, default=-30)
numDisparities_slider = create_slider("numDisparities", 16, 256, 16, 160)
preFilterSize_slider = create_slider("preFilterSize (odd)", 5, 255, 2, 5)
textureThreshold_slider = create_slider("textureThreshold", 0, 1000, default=100)
uniquenessRatio_slider = create_slider("uniquenessRatio", 0, 100, default=10)
speckleRange_slider = create_slider("speckleRange", 0, 100, default=14)
speckleWindowSize_slider = create_slider("speckleWindowSize", 0, 200, default=100)
P1_slider = create_slider("P1", 0, 1000, default=216)
P2_slider = create_slider("P2", 0, 1000, default=864)
disp12MaxDiff_slider = create_slider("disp12MaxDiff", -1, 255, default=1)
# mode_slider = create_slider("mode", 0, 3, default=3)
```

10. Disparity Label and Canvas:
    - A label (`disparity_label`) is created to display the disparity value when the user clicks on the map.
    - A canvas (`disparity_canvas`) is created to display the disparity map. It occupies the remaining space in the window and can expand if the window is resized.

```
# Disparity Label
disparity_label = tk.Label(slider_frame, text="Disparity:")
disparity_label.grid(row=slider_row, column=0, sticky="w")

# Disparity Canvas
disparity_canvas = tk.Canvas(window)
disparity_canvas.grid(row=0, column=1, sticky="nsew")
```

11. Load Images and Initialize:
    - Attempts to load the stereo images and calls the `update_disparity_map` function to calculate and display the initial disparity map.
    - If any error occurs during loading or calculation, it displays an error message in the console and the GUI.

```
# Load images and initialize the disparity map on startup
try:
    left_image = cv2.imread("ImageL_1.bmp", cv2.IMREAD_GRAYSCALE)
    right_image = cv2.imread("ImageR_1.bmp", cv2.IMREAD_GRAYSCALE)
    update_disparity_map()  # Initialize and display the disparity map.
except Exception as e:
    print(f"Error: {e}")  # Print the error message
    disparity_label.config(text=f"Error: {e}")  # Display error in GUI
```

12. Configure Layout and Start Main Loop:
    - Configures the grid layout to allow the disparity canvas to expand when the window is resized.
    - Starts the Tkinter main event loop, which keeps the window open and responds to user events (like slider changes or mouse clicks).

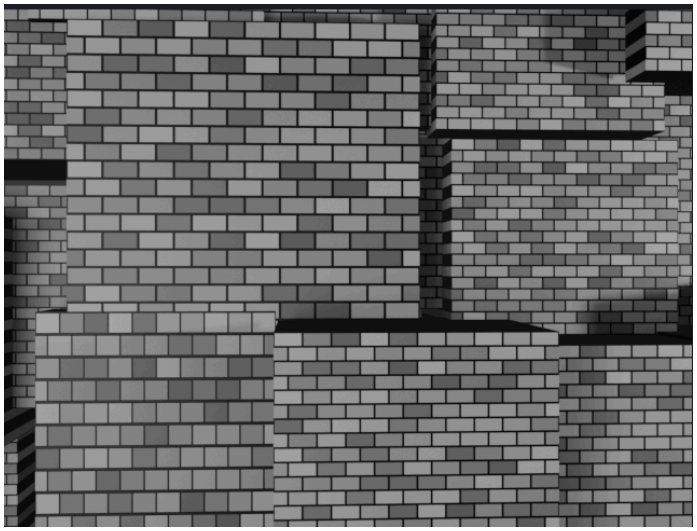
```
# Configure grid layout
window.grid_rowconfigure(0, weight=1)  # Allow image canvas to expand vertically
window.grid_columnconfigure(1, weight=1)  # Allow image canvas to expand horizontally

window.mainloop()
```

Key Connections

- **Camera Control:** The camera acquisition and setup are identical to the first script, ensuring seamless integration.
- **Disparity Map:** The disparity map calculation in disp.disparityCalculate likely uses the SGBM algorithm, similar to the second script.
- **GUI Integration:** The disp.createTrackbars function suggests that the disparity map visualization and depth measurement are done through an interactive GUI, like the third script.
- **Depth Measurement:** The onMouse function allows the user to click on the disparity map to get the distance of the corresponding point, making the system more interactive.

OBSERVATIONS & RECORDINGS :



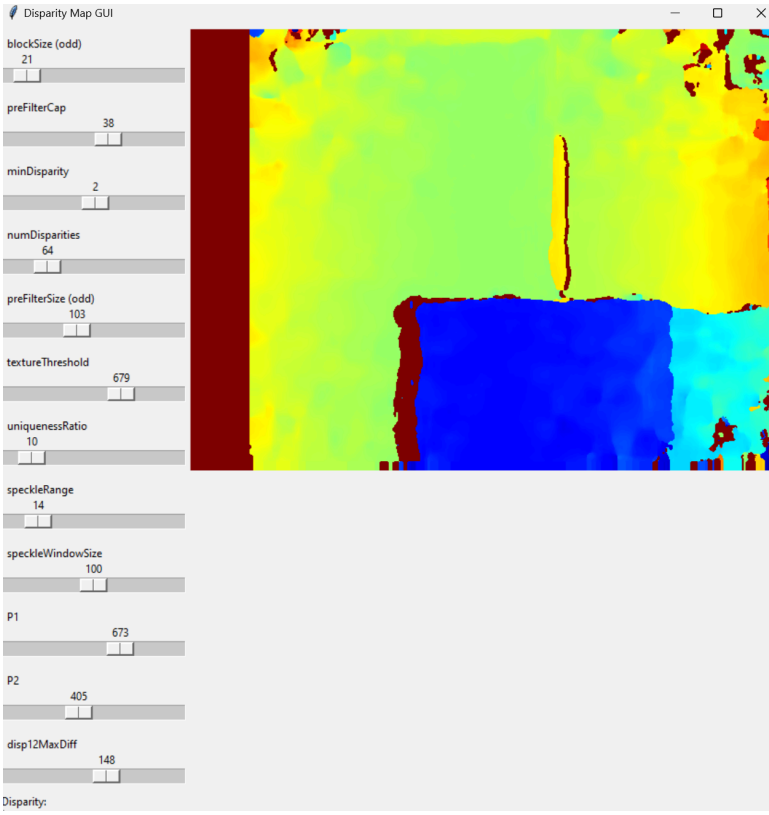
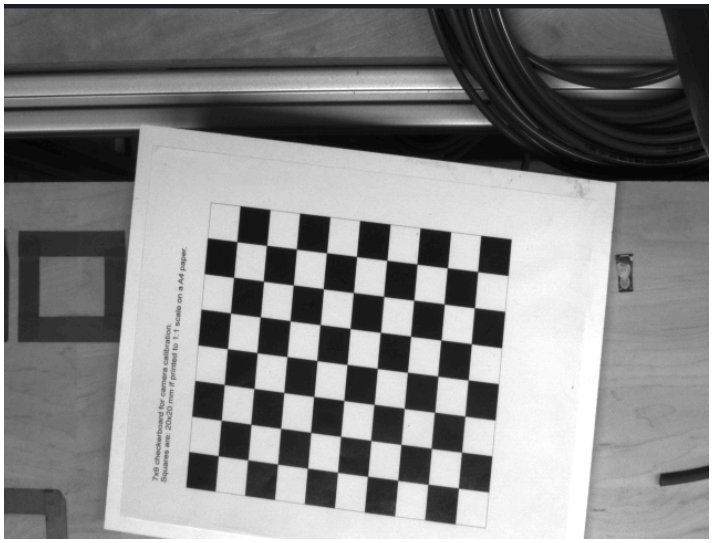


Fig. 3: Figure Generated through Blendr (Above), Disparity MAP (Right)



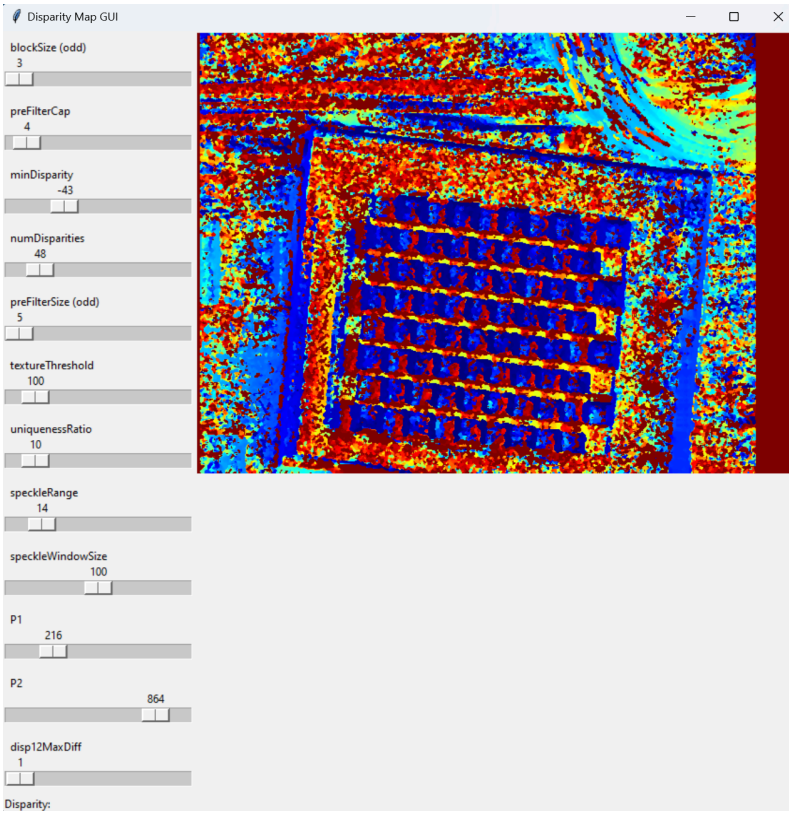
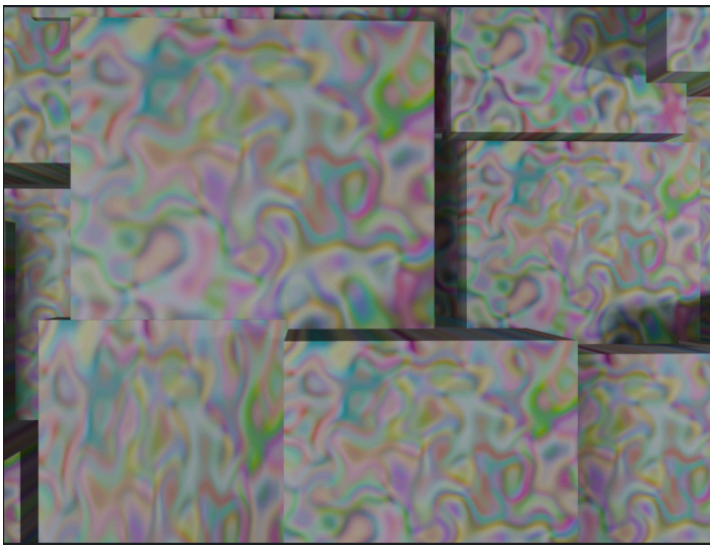


Fig. 4: Photo captured using Camera (Above), Disparity MAP (Right)



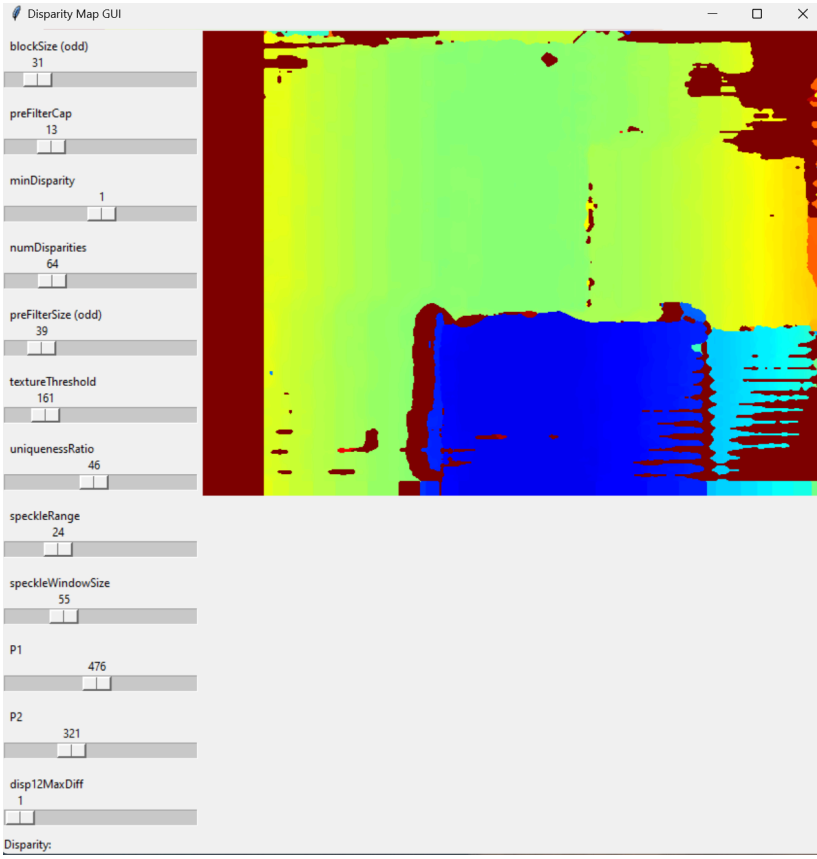
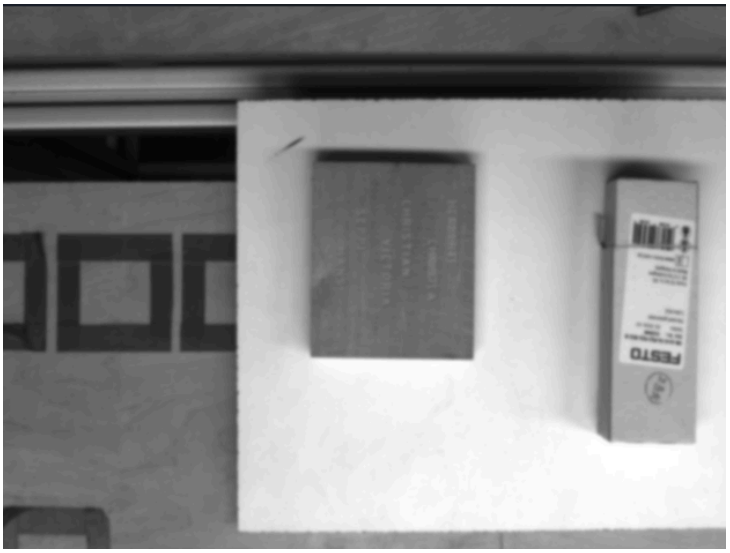


Fig. 5: Figure Generated through Blendr (Above), Disparity MAPI (Right)



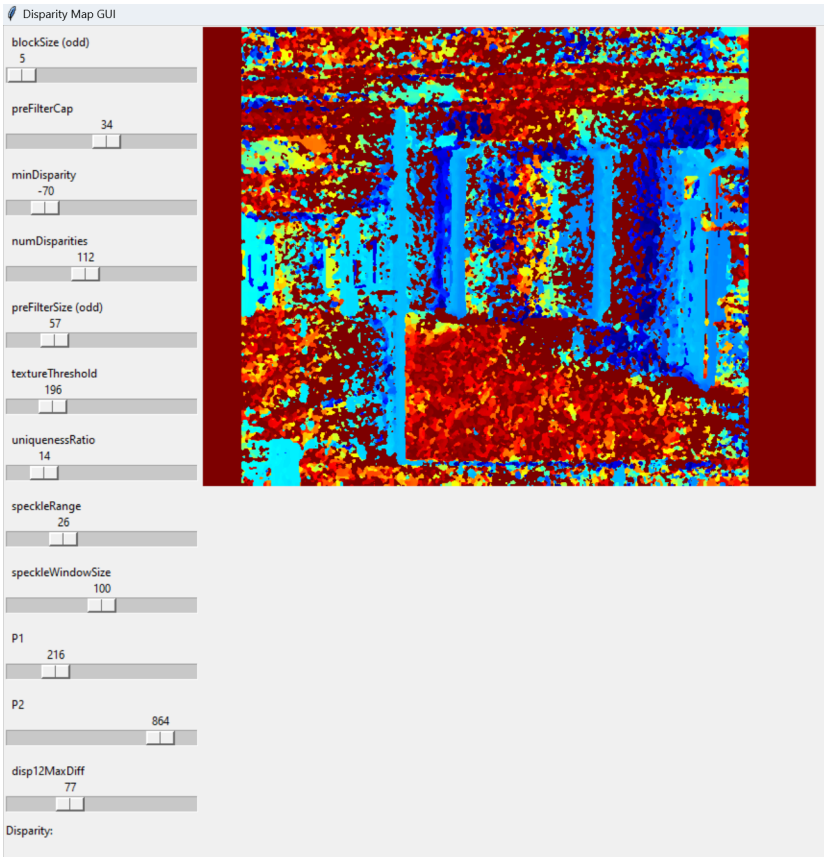
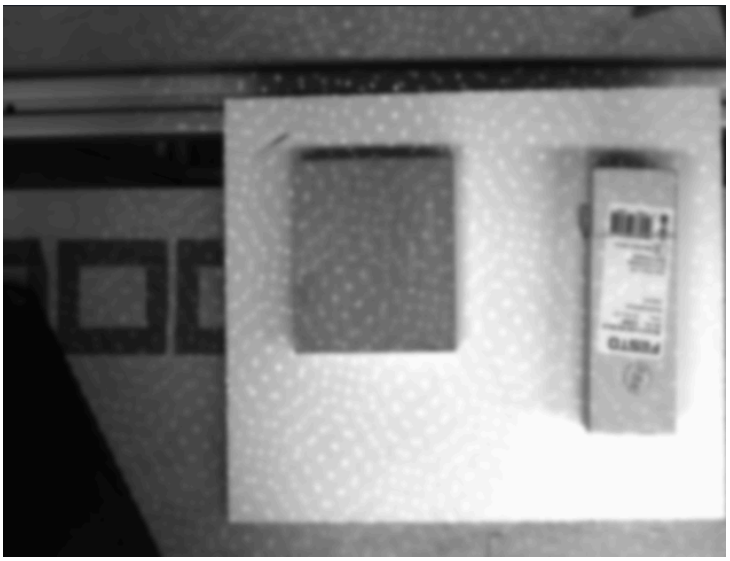


Fig. 6:Smoothened image using ICONet by applying Gaussian Blur (Above) , Disparity MAP (Right)



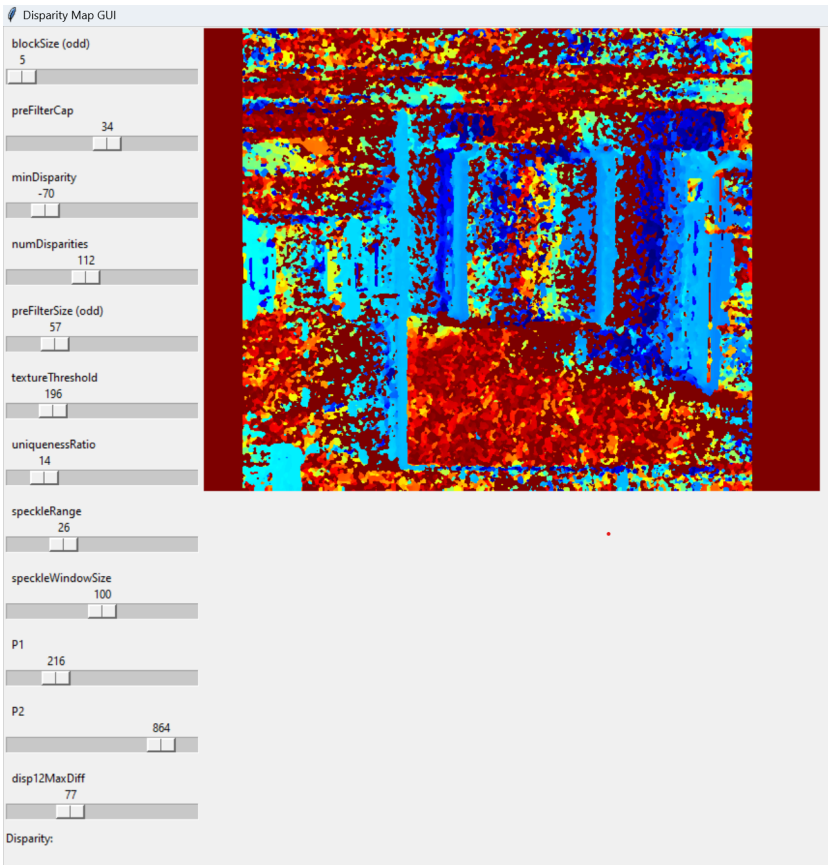
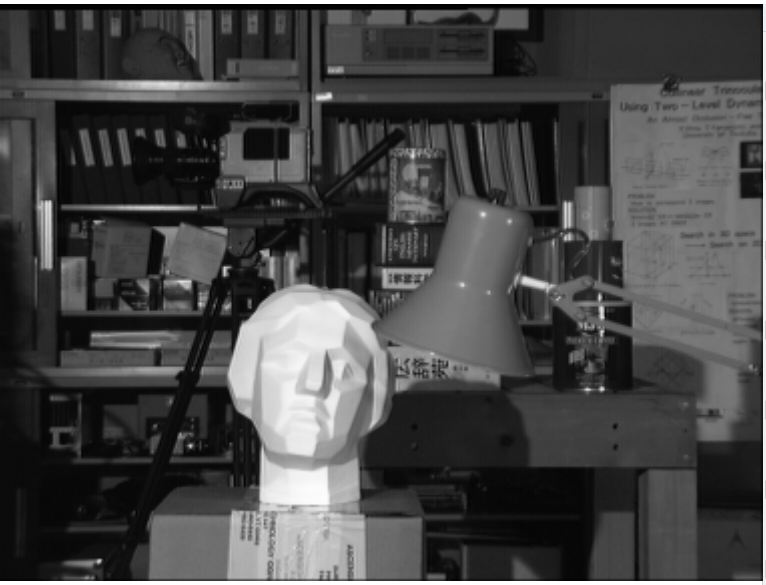


Fig. 7: Externally Smoothened Image (Above)
Disparity Map(Right)



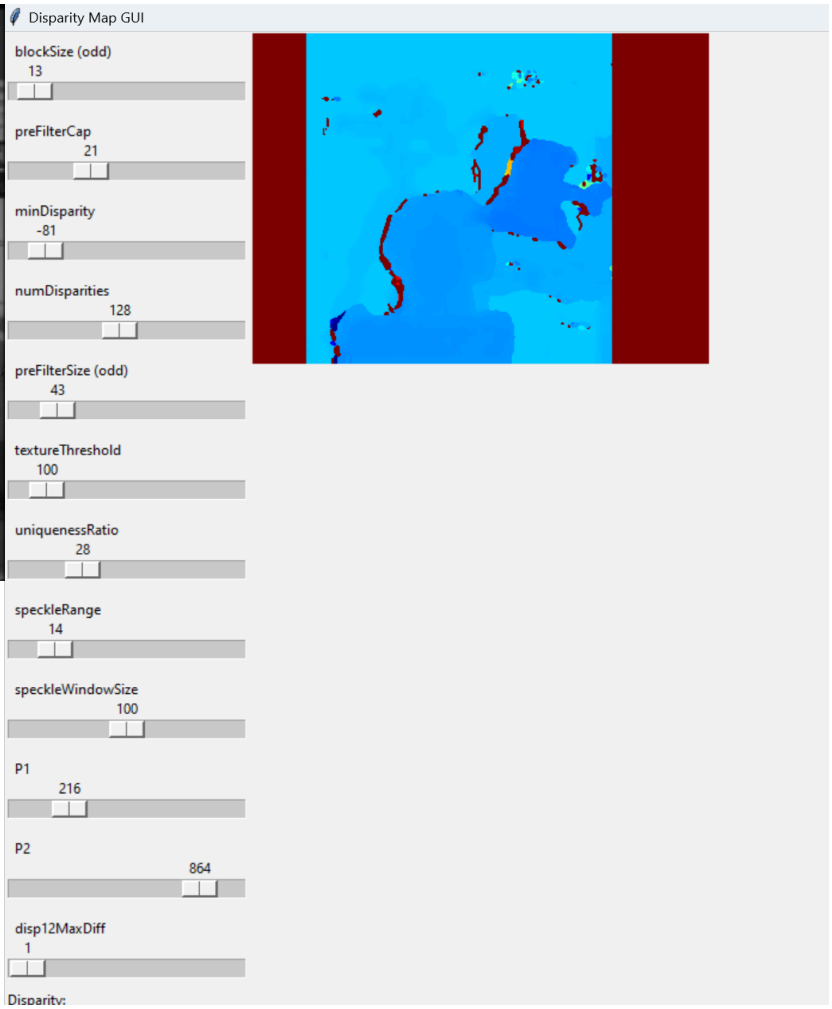


Fig. 8: Grayscale Image(Above),Disparity MAP(RIght)

C Conclusion

The successful development of this stereo calibration system underscores the efficacy of computer vision techniques in achieving accurate depth perception. The meticulous calibration process, outlined in this documentation, empowers researchers and developers to confidently embark on projects that harness the power of stereo vision.

By leveraging the calibrated parameters and rectification maps obtained through this work, the potential for advancements in diverse fields becomes evident. Real-time disparity calculation can enable robots to navigate complex environments, while precise 3D reconstruction can enhance augmented reality experiences. The system's flexibility facilitates further customization and experimentation, enabling the exploration of advanced algorithms such as semi-global matching (SGBM) or dense matching to enhance depth accuracy.

It is anticipated that continued research and refinement of this calibration framework will pave the way for innovative advancements in various domains, including robotics, autonomous systems, medical imaging, and beyond. The insights and resources provided herein equip researchers and developers with a robust foundation to contribute to the ongoing evolution of stereo vision technology.

Appendix

The libraries utilised in this code must be pre installed using the command line using ‘pip’ command:

- Open Command prompt as an admin.
- Type the command: python -m pip install numpy
- Similarly install the other required libraries.

Vimba Camera Setup:

```
import cv2 as cv
import numpy as np
from vmbpy import *
import time
from queue import Queue
from typing import Optional, Callable
import matplotlib.pyplot as plt

# Import custom modules for calibration and disparity calculations
import Calibration
import Disparity
import DISPARITY as disp

# --- Callback Functions ---

def onMouse(event, x, y, flags, disparityNormalized):
    """
    Handles mouse click events on the disparity map.
    Calculates and prints the distance in centimeters at the clicked pixel.
    """
    if event == cv.EVENT_LBUTTONDOWN:
        distance = disparityNormalized[y][x]
        print(f"Distance in Centimeters: {distance}")
        return distance
```

```
# --- Camera Setup and Configuration ---

opencv_display_format = PixelFormat.Bgr8  # Preferred pixel format for OpenCV display

def get_camera():
    """
    Retrieves all available cameras from the Vimba system.
    Raises an error if no cameras are found.
    """
    with VmbSystem.get_instance() as vmb:
        cameras = vmb.get_all_cameras()
        if not cameras:
            raise RuntimeError("No cameras detected")
        return cameras

def setup_camera(cam: Camera):
    """
    Configures common camera settings:
    - Sets exposure to automatic .
    - Sets white balance to automatic .
    - Adjusts packet size for optimal streaming .
    """
    with cam:
        try:
            cam.ExposureAuto.set("Continuous")
            cam.BalanceWhiteAuto.set("Continuous")

            stream = cam.get_stream()[0]  # Assuming one stream per camera
            stream.GVSPAdjustPacketSize.run()
            while not stream.GVSPAdjustPacketSize.is_done():  # Wait for adjustment to complete
                pass
        except (AttributeError, VmbFeatureError) as e:
            print(f"Warning: Failed to set up camera feature: {e}")  # Log any issues


class Handler:
    """
    Handles incoming frames from the camera stream.
    - Queues frames for display in the correct format.
    """
    def __init__(self):
        self.display_queue = Queue(10)

    def get_image(self):
        """Retrieves the next image from the queue (blocking)."""
        return self.display_queue.get(True)
```

```
    def __call__(self, cam: Camera, stream: Stream, frame: Frame):
        """
        Callback function called for each frame received from the camera.
        Converts the frame to the desired format and puts it in the display queue.
        """
        if frame.get_status() == FrameStatus.Complete:
            print(f"{cam} acquired {frame}", flush=True)

            if frame.get_pixel_format() != opencv_display_format:
                frame = frame.convert_pixel_format(opencv_display_format)

            self.display_queue.put(frame.as_opencv_image(), True)
        cam.queue_frame(frame)  # Re-queue the frame



# --- Trigger Functions  ---

def triggerCamera(camera):
    """Configures camera for software triggering (implementation not shown)."""
    pass

def trigger(camera):
    """Triggers the camera to capture a frame (implementation not shown)."""
    pass



# --- Main Program ---

with VmbSystem.get_instance() as vmb:

    # Get and setup the first two cameras
    cameras = get_camera()
    if len(cameras) < 2:
        raise RuntimeError("Need at least two cameras for stereo vision")

    with cameras[0] as camL, cameras[1] as camR:
        setup_camera(camR)
        setup_camera(camL)

        # Configure triggering and set manual exposure
        triggerCamera(camL)
        triggerCamera(camR)
        camR.ExposureAuto.set("Off")
        camR.ExposureTime.set(20000)
        camL.ExposureAuto.set("Off")
        camL.ExposureTime.set(20000)
```

```
# ... (Load default camera settings if needed)

# Start video streaming with handlers
handlerR = Handler()
handlerL = Handler()
camR.start_streaming(handler=handlerR, buffer_count=10)
camL.start_streaming(handler=handlerL, buffer_count=10)

# Display loop
msg = "Stream from '{}'. Press 'q' to quit, 's' to save images"
counter = 0
size = (640, 480)  # Desired display size
while True:
    # Get images from the queues
    imageR = cv.resize(handlerR.get_image(), size)
    imageL = cv.resize(handlerL.get_image(), size)

    grayR = cv.cvtColor(imageR, cv.COLOR_BGR2GRAY)
    grayL = cv.cvtColor(imageL, cv.COLOR_BGR2GRAY)

    # Display the images
    cv.imshow(msg.format(camR.get_name()), grayR)  # Right camera
    cv.imshow(msg.format(camL.get_name()), grayL)  # Left camera

    key = cv.waitKey(1)
    if key == ord("q"):  # Quit on 'q'
        break
    elif key == ord("s"):  # Save images on 's'
        print("Images Saved!!!")
        counter += 1
        # ... (Save the images to desired location)

# Cleanup on exit
cv.destroyAllWindows()
camR.stop_streaming()
camL.stop_streaming()
```

Stereo Calibration:

```
import cv2 as cv
import numpy as np
import glob

def StereoCalibration():
    ########## FIND CHESSBOARD CORNERS ########################
    boardsize = (9,7)
    framesize = (640,480)

    # termination criteria
    criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

    #prepare object points with numpy
    objP = np.zeros((boardsize[0]*boardsize[1],3), np.float32)
    objP[:,:2] = np.mgrid[0:boardsize[0],0:boardsize[1]].T.reshape(-1,2)

    # Arrays to store object points and image points from all the images.
    objPoints = [] # 3d point in real world space
    imgPointsL = [] # 2d points in image plane of camera1.
    imgPointsR = [] # 2d points in image plane of camera0.

    imagesL = glob.glob("S:\Bin Picking Project\TEST\CamL\*.bmp")
    imagesR = glob.glob("S:\Bin Picking Project\TEST\CamR\*.bmp")


    for imageL, imageR in zip(imagesL, imagesR):

        imgL = cv.imread(imageL)
        imgR = cv.imread(imageR)

        grayL = cv.cvtColor(imgL, cv.COLOR_BGR2GRAY)
        grayR = cv.cvtColor(imgR, cv.COLOR_BGR2GRAY)


        #Find corners in the chessboard
        retL, cornersL = cv.findChessboardCorners(grayL, boardsize, None)
        retR, cornersR = cv.findChessboardCorners(grayR, boardsize, None)

        print(retL, retR)
        # If corners are found, add object points and image points
        if retL and retR == True:
            objPoints.append(objP)
```

```
        #append the chessboard corner point to imagepoints
        cornersL = cv.cornerSubPix(grayL, cornersL, (11,11),(-1,-1), criteria)
        imgPointsL.append(cornersL)
        cornersR = cv.cornerSubPix(grayR, cornersR, (11,11),(-1,-1), criteria)
        imgPointsR.append(cornersR)


        # draw and display the corners
        cv.drawChessboardCorners(imgL, boardsize,cornersL, retL)
        cv.imshow("LEFT_img", imgL)
        cv.drawChessboardCorners(imgR, boardsize,cornersR, retR)
        cv.imshow("RIGHT_img", imgR)


        cv.waitKey(3000)

    cv.destroyAllWindows()


    ############CALIBRATION ################################################
    #Camera1 Calibration
                    calibration1L, cameraMatrixL, distortionL, rotVectorL, transVectorL =
cv.calibrateCamera(objPoints, imgPointsL, grayL.shape[::-1], None, None)
    heightL, widthL, channelsL = imgL.shape
    #  # #### Returns the new camera intrinsic matrix based on the free scaling parameter.
                    newCameraMatrixL, roi_L = cv.getOptimalNewCameraMatrix(cameraMatrixL,
distortionL,(widthL, heightL), 1, (widthL, heightL))

    # Camera2 Calibration
                    calibrationR, cameraMatrixR, distortionR, rotVectorR, transVectorR =
cv.calibrateCamera(objPoints, imgPointsR, grayR.shape[::-1], None, None)
    heightR, widthR, channelsR = imgR.shape
    #  # #### Returns the new camera intrinsic matrix based on the free scaling parameter.
                    newCameraMatrixR, roi_R = cv.getOptimalNewCameraMatrix(cameraMatrixR,
distortionR,(widthR, heightR), 1, (widthR, heightR))

    ################ Stereo Vision Calibration #########################
    flags = 0
    flags |= cv.CALIB_FIX_INTRINSIC
      # here we fix the intrinsic camera matrices so that only Rotation, translation essential and
fundamental matrices
    # so the intrinsic parameters are the same
```

```
ret, newCameraMatrixL, distCoeffsL, newCameraMatrixR, distCoeffsR, rotMatrix, transVector, essMatrix, fundMatrix = cv.stereoCalibrate(objPoints, imgPointsL, imgPointsR, newCameraMatrixL, distortionL,
                                                                                                newCameraMatrixR, distortionR, grayR.shape[::-1], criteria=criteria, flags=flags)



    ############# Stereo Rectification #########################################
    ##  Computes rectification transforms for each head of a calibrated stereo camera
    ### alpha = 1
    ## #alpha=0 means that the rectified images are zoomed and shifted
    rectifyScale = 1
    rectTransL, rectTransR, projMatrixL, projMatrixR, disp2DepthMapMatrix, roiL, roiR = cv.stereoRectify(newCameraMatrixL, distCoeffsL, newCameraMatrixR, distCoeffsR, grayR.shape[::-1], rotMatrix, transVector, rectifyScale, (0,0))


    stereoMapLx, stereoMapLy = cv.initUndistortRectifyMap(cameraMatrixL, distCoeffsL, rotMatrix, projMatrixL, grayL.shape[::-1], cv.CV_32FC1)

    stereoMapRx, stereoMapRy = cv.initUndistortRectifyMap(cameraMatrixR, distCoeffsR, rotMatrix, projMatrixR, grayR.shape[::-1], cv.CV_32FC1)

    print("Saving Calibration Parameters")
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\cameraMatrixLeft.npy', cameraMatrixL)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\cameraMatrixRight.npy', cameraMatrixR)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\newcameraMatrixLeft.npy', newCameraMatrixL)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\newcameraMatrixRight.npy', newCameraMatrixR)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\Distortion_CoefficentLeft.npy', distCoeffsL)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\Distortion_CoefficentRight.npy', distCoeffsR)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\RotationMatrixLR.npy', rotMatrix)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\TranslationVectorLR.npy', transVector)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\projectionMatrixLeft.npy', projMatrixL)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\projectionMatrixRight.npy', projMatrixR)
    np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\rotationMatrixLeft.npy', rectTransL)
```

```
        np.save('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\rotationMatrixRight.npy', rectTransR)

np.save('S:\Bin PickingProject\TEST\pixel2WCos\calibParams\disparity2depthMapMatrix.npy',disp2DepthMapMatrix)
    print("saving Parameters")
     cv_file = cv.FileStorage('S:\Bin Picking Project\TEST\pixel2WCos\calibParams\stereoMap.xml', cv.FILE_STORAGE_WRITE)

    cv_file.write('stereoMapL_x', stereoMapLx)
    cv_file.write('stereoMapL_y', stereoMapLy)
    cv_file.write('stereoMapR_x', stereoMapRx)
    cv_file.write('stereoMapR_y', stereoMapRy)

    cv_file.release()

StereoCalibration()
```

Stereo Rectification:

```
import sys
import cv2 as cv
import numpy as np

# --- Load Calibration Parameters ---
camMatrix0= =
np.load('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/cameraMatrixLeft.npy')
camMatrix1= =
np.load('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/cameraMatrixRight.npy')
distcoef0= =
np.load('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/Distortion_CoefficentLeft.np
y')
distcoef1= =
np.load('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/Distortion_CoefficentRight.n
py')
newCamMat0= =
np.load('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/newcameraMatrixLeft.npy')
newCamMat1= =
np.load('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/newcameraMatrixRight.npy')
rectTransL= =
np.load('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/rotationMatrixLeft.npy')
rectTransR= =
np.load('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/rotationMatrixRight.npy')
projMatrixL= =
np.load('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/projectionMatrixLeft.npy')
projMatrixR= =
np.load('C:/Users/localuser/Documents/test/pixel2WCos/calibParams/projectionMatrixRight.npy')


# --- Image Undistortion and Rectification Functions ---

def undistort(image0, image1):
    """
    Undistorts a pair of stereo images based on loaded camera parameters.

    This function applies lens distortion correction to both input images using the
    calibration matrices (`camMatrix0`, `camMatrix1`) and distortion coefficients
    (`distcoef0`, `distcoef1`) obtained during calibration.
    The new camera matrices (`newCamMat0`, `newCamMat1`) might be used for optimal
    undistortion if they were computed during calibration.

    Args:
        image0: The left image (numpy array).
        image1: The right image (numpy array).
```

```
    Returns:
        undistorted0: The undistorted left image.
        undistorted1: The undistorted right image.
    """
    undistorted0 = cv.undistort(image0, camMatrix0, distcoef0, None, newCamMat0)
    undistorted1 = cv.undistort(image1, camMatrix1, distcoef1, None, newCamMat1)
    return undistorted0, undistorted1


def undistortRectify(imageL, imageR):
    """
    Undistorts and rectifies a pair of stereo images.

    This function first calculates rectification maps based on the loaded calibration parameters.
    It then uses these maps to undistort and rectify the input images, making the epipolar
    lines parallel for easier stereo correspondence calculations.

    Args:
        imageL: The left image (numpy array).
        imageR: The right image (numpy array).

    Returns:
        undistortedL: The undistorted and rectified left image.
        undistortedR: The undistorted and rectified right image.
    """

    # Initialize rectification maps
    stereoMapL_x, stereoMapL_y = cv.initUndistortRectifyMap(
        camMatrix0, distcoef0, rectTransL, projMatrixL, (640, 480), cv.CV_32FC1
    )
    stereoMapR_x, stereoMapR_y = cv.initUndistortRectifyMap(
        camMatrix1, distcoef1, rectTransR, projMatrixR, (640, 480), cv.CV_32FC1
    )

    # Undistort and rectify images using the maps
    undistortedL = cv.remap(imageL, stereoMapL_x, stereoMapL_y, cv.INTER_LINEAR)
    undistortedR = cv.remap(imageR, stereoMapR_x, stereoMapR_y, cv.INTER_LINEAR)

    return undistortedL, undistortedR
```

Disparity:

```
import cv2 as cv
import numpy as np
import glob
import sys
import matplotlib.pyplot as plt

# --- Calibration Parameters ---
boardSize = (7, 9)  # Inner corners on chessboard (horizontal, vertical)
frameSize = (640, 480)  # Image resolution (width, height)
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)  # Termination criteria for corner refinement

# --- 3D Object Points (Chessboard Corners) ---
objp = np.zeros((boardSize[0] * boardSize[1], 3), np.float32)  # Create a grid of 3D points (z=0 for flat chessboard)
objp[:, :2] = np.mgrid[0:boardSize[0], 0:boardSize[1]].T.reshape(-1, 2)  # Fill with corner coordinates

# --- Storage for Calibration Data ---
objPoints = []  # 3D points in real world space
imgPointsL = []  # 2D points in left image plane
imgPointsR = []  # 2D points in right image plane

# --- Function: Camera Calibration ---
def calibrate_camera(images, boardSize, frameSize, criteria):
    """
    Calibrates a single camera using a set of chessboard images.

    Args:
        images: List of image file paths.
        boardSize: Size of the chessboard (inner corners).
        frameSize: Image resolution (width, height).
        criteria: Termination criteria for corner refinement.

    Returns:
        ret: Calibration success flag.
        mtx: Camera matrix (intrinsic parameters).
        dist: Distortion coefficients.
        rvecs: Rotation vectors (extrinsic parameters).
        tvecs: Translation vectors (extrinsic parameters).
    """
    objpoints = []
```

```
    imgpoints = []

    for image in images:
        img = cv.imread(image)
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

        # Find chessboard corners
        ret, corners = cv.findChessboardCorners(gray, boardSize, None)

        if ret:
            # Refine corner locations to subpixel accuracy
            corners2 = cv.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)

            # Store object points and image points
            objpoints.append(objp)
            imgpoints.append(corners2)

            # Visualize corners (optional for debugging)
            img = cv.drawChessboardCorners(img, boardSize, corners2, ret)
            cv.imshow('img', img)
            cv.waitKey(500)

    cv.destroyAllWindows()

    # Calibrate camera
    ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, frameSize, None, None)
    return ret, mtx, dist, rvecs, tvecs

# --- Function: Disparity Map and Depth Estimation ---
def disparityMap(imageL, imageR):
    """
    Calculates the disparity map and depth information from a pair of rectified stereo images.

    Args:
        imageL: Rectified left image.
        imageR: Rectified right image.

    Returns:
        disparity: Disparity map (pixel differences between corresponding points).
        depth: Depth map (estimated distances to scene points).
    """
    # Create StereoBM object and set parameters
    ndispFactor = 18  # Disparity search range multiplier
    stereo = cv.StereoBM_create(numDisparities=0 * ndispFactor, blockSize=21)
```

```
    stereo.setTextureThreshold(10)  # Filter out areas with low texture
    stereo.setSpeckleRange(1000)  # Reduce speckles in the disparity map

    # Compute disparity
    disparity = stereo.compute(imageL, imageR)

    # Load pre-computed disparity-to-depth transformation matrix
disp2depthmapMat = np.load('S:\Bin Picking Project\TEST\pixel2WCos\disparity2depthMapMatrix.npy')

    # Normalize and visualize disparity (optional for debugging)
     disp_norm = cv.normalize(disparity, None, alpha=0, beta=255, norm_type=cv.NORM_MINMAX, dtype=cv.CV_8U)
    disp_norm = cv.applyColorMap(disp_norm, cv.COLORMAP_JET)

    # Calculate depth map
    depth = cv.reprojectImageTo3D(disparity, disp2depthmapMat)

    return disparity, depth

# --- Function: Mouse Callback for Depth Measurement
def onMouse(event, x, y, flag, disparityNormalized):
    """
    Handles mouse events to display depth information at the clicked point.
    """
    if event == cv.EVENT_LBUTTONDOWN:
        distance = disparityNormalized[y][x]
        print("Distance in Centimeters:", distance)
        return distance


# --- Function: Enhanced Depth Map Calculation with StereoSGBM and WLS Filtering ---
def depth_map(imgL, imgR, Q, leftMapX, leftMapY, rightMapX, rightMapY):
    """
       Calculates a refined depth map using StereoSGBM and Weighted Least Squares (WLS) filtering.

    Args:
        imgL: Rectified left image.
        imgR: Rectified right image.
        Q: Disparity-to-depth mapping matrix (from stereoRectify).
        leftMapX, leftMapY: Remapping maps for the left image.
        rightMapX, rightMapY: Remapping maps for the right image.

    Returns:
```

```
    filteredImg: Filtered disparity map (for visualization).
    points_3D: 3D point cloud in the camera coordinate system.
  """
  # ... (SGBM parameters and setup, image remapping, disparity calculation, WLS filtering, etc.)

  def depth_map(imgL, imgR, Q, leftMapX, leftMapY, rightMapX, rightMapY):
  window_size = 3  # SGBM Parameters
  left_matcher = cv.StereoSGBM_create(
    minDisparity=-1,
    numDisparities=5*16,
    blockSize=window_size,
    P1=8 * 3 * window_size,
    P2=32 * 3 * window_size,
    disp12MaxDiff=12,
    uniquenessRatio=10,
    speckleWindowSize=50,
    speckleRange=32,
    preFilterCap=63,
    mode=cv.STEREO_SGBM_MODE_SGBM_3WAY
  )
  right_matcher = cv.ximgproc.createRightMatcher(left_matcher)
  lmbda = 80000
  sigma = 1.3

  wls_filter = cv.ximgproc.createDisparityWLSFilter(matcher_left=left_matcher)
  wls_filter.setLambda(lmbda)
  wls_filter.setSigmaColor(sigma)

  # Rectify Images
  imgL = cv.remap(imgL, leftMapX, leftMapY, cv.INTER_LINEAR)
  imgR = cv.remap(imgR, rightMapX, rightMapY, cv.INTER_LINEAR)

  # Disparity Calculation and Filtering
  disparity = left_matcher.compute(imgL, imgR)
  disparity = np.int16(disparity)
  filteredImg = wls_filter.filter(disparity, imgL, None, None)

  # Convert to Depth Map
  points_3D = cv.reprojectImageTo3D(filteredImg, Q)

  # Normalize for Visualization
        filteredImg = cv.normalize(src=filteredImg, dst=filteredImg, beta=0, alpha=255,
norm_type=cv.NORM_MINMAX)
  filteredImg = np.uint8(filteredImg)
```

```
    return filteredImg, points_3D

# --- Main Program ---
if __name__ == "__main__":
    # Check if image paths were provided as arguments
    if len(sys.argv) < 3:
        print("Usage: python your_script.py <left_image_path> <right_image_path>")
        sys.exit(1)

    left_image_path = sys.argv[1]
    right_image_path = sys.argv[2]

    # Load and Preprocess Images (After Calibration)
    imagesR = cv.imread(right_image_path)
    imagesL = cv.imread(left_image_path)

    if imagesR is None or imagesL is None:
        raise ValueError("Error loading images. Check file paths and ensure images are valid.")

    imageSize = imagesL.shape[:2]

    # Load Images (For Both Cameras)
    imagesL_cali = glob.glob(r"S:\Bin Picking Project\TEST\CamL\*.bmp")
    imagesR_cali = glob.glob(r"S:\Bin Picking Project\TEST\CamR\*.bmp")

    # Camera Calibration
        retL, mtxL, distL, rvecsL, tvecsL = calibrate_camera(imagesL_cali, boardSize, frameSize,
criteria)
      retR, mtxR, distR, rvecsR, tvecsR = calibrate_camera(imagesR_cali, boardSize, frameSize,
criteria)

    # Check if there are enough points for calibration
    if len(objPoints) < 2:
        print("Not enough calibration images found. Please use at least 2 images.")
        exit()

    # Stereo Calibration
    flags = 0
    flags |= cv.CALIB_FIX_INTRINSIC
    criteria_stereo = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

        retS, newCameraMatrixL, distCoeffsL, newCameraMatrixR, distCoeffsR, R, T, E, F =
cv.stereoCalibrate(
        objPoints, imgPointsL, imgPointsR, mtxL, distL, mtxR, distR,
```

```
        imageSize, criteria_stereo, flags
    )
    if not retS:
        raise ValueError("Stereo calibration failed. Check your calibration images and settings.")

    # Stereo Rectification
    rectify_scale = 1
    R1, R2, P1, P2, Q, validPixROI1, validPixROI2 = cv.stereoRectify(
        newCameraMatrixL, distCoeffsL, newCameraMatrixR, distCoeffsR,
        imageSize, R, T, rectify_scale, (0, 0)
    )
        left_map1, left_map2 = cv.initUndistortRectifyMap(newCameraMatrixL, distCoeffsL, R1, P1, imageSize, cv.CV_16SC2)
     right_map1, right_map2 = cv.initUndistortRectifyMap(newCameraMatrixR, distCoeffsR, R2, P2, imageSize, cv.CV_16SC2)


    # Calculate Disparity and Depth Map (Pass rectification maps)
       disparity, points_3D = depth_map(imagesL, imagesR, Q, left_map1, left_map2, right_map1, right_map2)

    # Visualization
    plt.imshow(disparity, 'gray')
    plt.title('Disparity Map')
    plt.show()
```

Disparity GUI:

```
import tkinter as tk #Library used for making GUI
import cv2
import numpy as np
from PIL import Image, ImageTk

# --- Image Loading and Error Handling ---

# Load stereo images (BMP format)
try:
    left_image = cv2.imread("ImageL_1.bmp", cv2.IMREAD_GRAYSCALE)
    right_image = cv2.imread("ImageR_1.bmp", cv2.IMREAD_GRAYSCALE)
except Exception as e:
    print(f"Error loading images: {e}")
    exit()  # Exit the program if images can't be loaded

# Validate image loading
if left_image is None or right_image is None:
    print("Error: Could not load one or both of the images.")
    exit()

# --- Disparity Map Initialization ---

disparity = None          # Initialize disparity map (will be calculated later)
disparity_color = None    # Initialize color-mapped disparity map (for visualization)

# Create StereoSGBM object for disparity calculation (SGBM_3WAY mode)
stereo = cv2.StereoSGBM_create(mode=3)  # Using StereoSGBM algorithm for depth estimation

# --- Mouse Callback Function ---

def on_mouse(event, x, y, flags, param):
    """
    Handles mouse events on the disparity map canvas.
    Displays the disparity value at the clicked point if valid.
    """
    global disparity
    if event == cv2.EVENT_LBUTTONDOWN and disparity is not None:
        # Check for valid coordinates
        if 0 <= y < disparity.shape[0] and 0 <= x < disparity.shape[1]:
            # Disparity values are scaled by 16, so divide by 16 for actual value
            disparity_value = disparity[y, x] / 16.0
```

```
            if disparity_value > 0:  # Ensure valid disparity
                disparity_label.config(text=f"Disparity: {disparity_value:.2f}")


# --- Disparity Map Update Function ---

def update_disparity_map(*args):
    """
    Updates the disparity map based on slider values and displays it on the canvas.
    """
    global disparity, disparity_color

    # --- Get and Validate Slider Values ---
    block_size = blockSize_slider.get()
    if block_size % 2 == 0:       # Ensure block size is odd
        block_size += 1
    preFilterSize = preFilterSize_slider.get()
    if preFilterSize % 2 == 0:     # Ensure pre-filter size is odd
        preFilterSize += 1

    # --- Set StereoSGBM Parameters ---
    stereo.setBlockSize(block_size)
    stereo.setPreFilterCap(preFilterCap_slider.get())
    stereo.setMinDisparity(minDisparity_slider.get())
    stereo.setNumDisparities(numDisparities_slider.get())
    textureThreshold = textureThreshold_slider.get()
    uniquenessRatio = uniquenessRatio_slider.get()
    speckleRange = speckleRange_slider.get()
    speckleWindowSize = speckleWindowSize_slider.get()
    P1 = P1_slider.get()
    P2 = P2_slider.get()
    disp12MaxDiff = disp12MaxDiff_slider.get()

    # --- Compute Disparity Map ---
    disparity = stereo.compute(left_image, right_image)

    # --- Normalize and Visualize ---
    # Normalize for 0-255 range for better visualization
            disparity_normalized = cv2.normalize(disparity, None, 0, 255, cv2.NORM_MINMAX,
cv2.CV_8UC1)
     disparity_color = cv2.applyColorMap(disparity_normalized, cv2.COLORMAP_JET)  # Apply JET
colormap

    # --- Display on Canvas ---
```

```
    disparity_image = Image.fromarray(disparity_color)
    disparity_photo = ImageTk.PhotoImage(disparity_image)

    # Update canvas dimensions to fit the image
    disparity_canvas.config(width=disparity_image.width, height=disparity_image.height)
    disparity_canvas.delete("all")  # Clear previous content
    disparity_canvas.create_image(0, 0, anchor=tk.NW, image=disparity_photo)

    # Keep a reference to avoid garbage collection of the image
    disparity_canvas.image = disparity_photo
    disparity_canvas.bind("<Button-1>", on_mouse)  # Enable mouse clicks for disparity display

# --- GUI Setup ---

window = tk.Tk()
window.title("Disparity Map GUI")

# Create a frame to hold all the sliders
slider_frame = tk.Frame(window)
slider_frame.grid(row=0, column=0, sticky="ns")  # Position the frame on the left side

# --- Slider Creation Helper Function ---

slider_row = 0  # Initialize the row counter for placing sliders

def create_slider(label, from_, to, resolution=1, default=None, length=200):
    """
    Helper function to create a slider widget with a label.

    Args:
        label: Text to display next to the slider.
        from_: Minimum value for the slider.
        to: Maximum value for the slider.
        resolution: Step size for the slider (default 1).
        default: Initial value of the slider (default None).
        length: Length of the slider (default 200).

    Returns:
        The created slider widget.
    """
    global slider_row  # Use the global row counter

    slider = tk.Scale(
        slider_frame,
```

```
        from_=from_,
        to=to,
        resolution=resolution,
        orient=tk.HORIZONTAL,  # Horizontal orientation
        label=label,
        length=length,
        command=update_disparity_map  # Call update_disparity_map on change
    )
    if default is not None:  # Set default value if provided
        slider.set(default)
     slider.grid(row=slider_row, column=0, sticky="ew", pady=5)  # Place in grid, expand horizontally,
add padding

    slider_row += 1  # Increment row for the next slider
    return slider

# --- Create Sliders ---

# Each slider controls a parameter of the StereoSGBM algorithm or its visualization
blockSize_slider = create_slider("Block Size (odd)", 3, 255, 2, 21)  # Block size for matching (odd
values only)
preFilterCap_slider = create_slider("Pre-Filter Cap", 1, 63, default=29)  # Pre-filtering threshold
minDisparity_slider = create_slider("Min Disparity", -100, 100, default=-30)  # Minimum possible
disparity value
numDisparities_slider = create_slider("Num Disparities", 16, 256, 16, 160)  # Number of disparity
values (multiple of 16)
preFilterSize_slider = create_slider("Pre-Filter Size (odd)", 5, 255, 2, 5)  # Pre-filtering window size
(odd values only)
textureThreshold_slider = create_slider("Texture Threshold", 0, 1000, default=100) # Threshold for
texture filtering
uniquenessRatio_slider = create_slider("Uniqueness Ratio", 0, 100, default=10)   # Uniqueness
ratio for matching
speckleRange_slider = create_slider("Speckle Range", 0, 100, default=14)          # Maximum
allowed difference between disparity values
speckleWindowSize_slider = create_slider("Speckle Window Size", 0, 200, default=100)   #
Window size for speckle removal
P1_slider = create_slider("P1", 0, 1000, default=216)  # Penalty for disparity change by +/- 1
between pixels
P2_slider = create_slider("P2", 0, 1000, default=864)  # Penalty for disparity change by more than
+/- 1 between pixels
disp12MaxDiff_slider = create_slider("Disp12 Max Diff", -1, 255, default=1)  # Maximum allowed
difference between left and right disparities


# --- Disparity Label ---
```

```
disparity_label = tk.Label(slider_frame, text="Disparity:")
disparity_label.grid(row=slider_row, column=0, sticky="w")

# --- Disparity Canvas ---
# Create a canvas to display the disparity map
disparity_canvas = tk.Canvas(window)
disparity_canvas.grid(row=0, column=1, sticky="nsew")  # Place on the right side, allow expansion

# --- Initial Disparity Map Display ---

# Load images and initialize the disparity map on startup
try:
    left_image = cv2.imread("ImageL_1.bmp", cv2.IMREAD_GRAYSCALE)
    right_image = cv2.imread("ImageR_1.bmp", cv2.IMREAD_GRAYSCALE)
    update_disparity_map()  # Initialize and display the disparity map
except Exception as e:
    print(f"Error: {e}")
    disparity_label.config(text=f"Error: {e}")  # Display error in GUI


# --- Grid Layout Configuration ---
# Make the image canvas expandable
window.grid_rowconfigure(0, weight=1)
window.grid_columnconfigure(1, weight=1)

window.mainloop()  # Start the GUI main loop
```

R References

Vimba Camera SDK and Python Bindings

- Vimba Python Manual (VmbPy):
  - Download: https://www.alliedvision.cn/fileadmin/content/documents/products/software/software/Vimba/docu/manuals/Vimba_Python_Manual.pdf
  - GitHub: https://github.com/alliedvision/VimbaPython/blob/master/Documentation/Vimba%20Python%20Manual.pdf

- Allied Vision Developer Guide (Python API Manual):
  - Online: https://docs.alliedvision.com/Vimba_DeveloperGuide/pythonAPIManual.html

OpenCV for Image Processing

- OpenCV Documentation:
  - Online: https://docs.opencv.org/4.x/index.html
- OpenCV-Python Tutorials:
  - Online: https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html
  - Online: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

Stereo Vision

- Stereo Vision Tutorial (OpenCV):
  - Online: https://docs.opencv.org/3.4/dd/d53/tutorial_py_depthmap.html

Disparity Calculation Algorithms

- Stereo Block Matching (StereoBM):
  - OpenCVDocumentation: https://docs.opencv.org/4.x/d9/dba/classcv_1_1StereoBM.html
- Semi-Global Block Matching (StereoSGBM):
  - OpenCVDocumentation: https://docs.opencv.org/4.x/d2/d85/classcv_1_1StereoSGBM.html
- Weighted Least Squares (WLS) Filter:
  - OpenCVDocumentation: https://docs.opencv.org/4.x/d3/d14/tutorial_ximgproc_disparity_filtering.html

GUI Frameworks

- Tkinter Documentation:
  - Online: https://docs.python.org/3/library/tkinter.html
- Matplotlib Documentation:
  - Online: https://matplotlib.org/stable/contents.html

Additional Resources

- Stack Overflow
- Allied Vision Support Forum: https://forum.alliedvision.com/