Calibration

Generated by Doxygen 1.12.0

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 BaslerCamera Class Reference

A class to interface with a Basler camera or emulate a camera using a video file.

`#include <BaslerCamera.hpp>`

Public Member Functions

- BaslerCamera (float exposure_time)

  Constructs a BaslerCamera object for a real camera with the specified exposure time.
- BaslerCamera (const std::string path_to_video_file)

  Constructs a BaslerCamera object to emulate a camera using a video file.
- ∼BaslerCamera ()

  Destructor for the BaslerCamera class.
- cv::Mat getFrame ()

  Retrieves the next frame from the camera or video file.

Public Attributes

- int camera_timeout = 5000

3.1.1 Detailed Description

A class to interface with a Basler camera or emulate a camera using a video file.

This class allows users to either interface with a physical Basler camera or emulate a camera by reading frames from a video file. It provides functionality to initialize the camera, retrieve frames, and handle resources properly.

Author

Anton Haes

3.1.2 Constructor & Destructor Documentation

3.1.2.1 BaslerCamera() [1/2]

```
BaslerCamera::BaslerCamera (
            float exposure_time)  [inline]
```

Constructs a BaslerCamera object for a real camera with the specified exposure time.

Initializes the Basler camera, sets it to continuous acquisition mode, and configures the exposure time.

Author

Anton Haes

Parameters

exposure_time	The exposure time for the camera in microseconds.

Exceptions

std::runtime_error	if the camera initialization fails.

3.1.2.2 BaslerCamera() [2/2]

```
BaslerCamera::BaslerCamera (
            const std::string path_to_video_file)  [inline]
```

Constructs a BaslerCamera object to emulate a camera using a video file.

Opens the specified video file for reading frames.

Author

Anton Haes

Parameters

path_to_video_file	The path to the video file to be used for emulation.

Exceptions

std::runtime_error	if the video file cannot be opened.

3.1.2.3 ∼BaslerCamera()

```
BaslerCamera::∼BaslerCamera ()  [inline]
```

Destructor for the BaslerCamera class.

Closes the camera and releases resources if the camera is not in emulation mode.

Author

Anton Haes

3.1.3 Member Function Documentation

3.1.3.1 getFrame()

```
cv::Mat BaslerCamera::getFrame ()  [inline]
```

Retrieves the next frame from the camera or video file.

If the camera is being emulated, it reads the next frame from the video file. If the camera is real, it grabs the latest frame from the camera.

Author

Anton Haes

Returns

cv::Mat The captured frame as an OpenCV Mat object. If the end of the video file is reached, it will return an empty frame.

Exceptions

std::runtime_error	if a frame cannot be retrieved or if the camera is not grabbing frames.

3.1.4 Member Data Documentation

3.1.4.1 camera_timeout

```
int BaslerCamera::camera_timeout = 5000
```

The documentation for this class was generated from the following file:

- include/BaslerCamera.hpp

3.2 Calibration Class Reference

Performs camera calibration using a series of chessboard patterns.

```
#include <Calibration.hpp>
```

Public Member Functions

- Calibration (BaslerCamera *cam)

    Constructs a Calibration object with the given BaslerCamera.
- ~Calibration ()
- void findChessboardCorners (Chessboard *chessboards[ ])

    Finds and records chessboard corners in images.
- void calibrate (Chessboard *chessboards[ ])

    Performs camera calibration using the detected chessboard corners.
- void save (const std::string &filename)

    Saves the calibration data to a file.
- void drawChessboardCorners (cv::Mat *frame, Chessboard *chessboards[ ])

    Draws detected chessboard corners on the provided image.

3.2.1 Detailed Description

Performs camera calibration using a series of chessboard patterns.

This class handles the process of camera calibration by finding chessboard corners in images, computing the camera's intrinsic and extrinsic parameters, and saving these parameters to a file. It also provides functionality to visualize the detected chessboard corners.

Author

Anton Haes

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Calibration()

```
Calibration::Calibration (
            BaslerCamera * cam)  [inline]
```

Constructs a Calibration object with the given BaslerCamera.

Author

Anton Haes

Parameters

cam	Pointer to a BaslerCamera object used to capture images for calibration.

3.2.2.2 ∼Calibration()

```
Calibration::∼Calibration ()  [inline]
```

3.2.3 Member Function Documentation

3.2.3.1 calibrate()

```
void Calibration::calibrate (
            Chessboard * chessboards[ ])  [inline]
```

Performs camera calibration using the detected chessboard corners.

This method computes the camera's calibration parameters based on the corners detected in the chessboard images. It constructs matrices for calibration, performs matrix decompositions, and computes the intrinsic and extrinsic parameters.

Author

Anton Haes

Parameters

chessboards	Array of pointers to Chessboard objects containing the detected corners and chessboard details.

3.2.3.2 drawChessboardCorners()

```
void Calibration::drawChessboardCorners (
            cv::Mat * frame,
            Chessboard * chessboards[ ])  [inline]
```

Draws detected chessboard corners on the provided image.

This method visualizes the detected chessboard corners by drawing circles around them on the given image frame.

Author

Anton Haes

Parameters

frame	Pointer to a cv::Mat object representing the image where the corners will be drawn.
chessboards	Array of pointers to Chessboard objects containing the detected corners.

3.2.3.3 findChessboardCorners()

```
void Calibration::findChessboardCorners (
            Chessboard * chessboards[ ])  [inline]
```

Finds and records chessboard corners in images.

This method iterates through an array of Chessboard objects and captures images from the camera to find chessboard corners within specified regions. It displays the frames with detected corners and adjusts corner coordinates based on the full frame.

Author

Anton Haes

Parameters

chessboards	Array of pointers to Chessboard objects, each representing a calibration pattern.

3.2.3.4 save()

```
void Calibration::save (
            const std::string & filename)  [inline]
```

Saves the calibration data to a file.

This method writes the camera calibration parameters, including the projection matrix, intrinsic matrix, rotation matrix, and translation vector, to a specified binary file. It also includes image dimensions.

Author

Anton Haes

Parameters

filename	The name of the file where calibration data will be saved.

Exceptions

std::runtime_error	If the file cannot be opened or created for writing.

The documentation for this class was generated from the following file:

- include/Calibration.hpp

3.3 CalibrationData Struct Reference

Contains data related to camera calibration.

```
#include <Calibration.hpp>
```

Public Attributes

- Eigen::MatrixXd P = Eigen::MatrixXd::Zero(3, 4)

  The projection matrix of the camera.
- Eigen::MatrixXd K_inv

  The inverse of the intrinsic matrix of the camera.
- Eigen::MatrixXd R_inv

  The inverse of the rotation matrix of the camera.
- Eigen::MatrixXd t

  The translation vector of the camera.
- int pixel_width = 1920

  Pixel width of the calibrated camera.
- int pixel_height = 1200

  Pixel height of the calibrated camera.

3.3.1 Detailed Description

Contains data related to camera calibration.

This structure holds the intrinsic and extrinsic parameters of the camera obtained from the calibration process. It includes the projection matrix, inverse intrinsic matrix, inverse rotation matrix, translation vector, and image dimensions.

Author

Anton Haes

3.3.2 Member Data Documentation

3.3.2.1 K_inv

```
Eigen::MatrixXd CalibrationData::K_inv
```

The inverse of the intrinsic matrix of the camera.

3.3.2.2 P

```
Eigen::MatrixXd CalibrationData::P = Eigen::MatrixXd::Zero(3, 4)
```

The projection matrix of the camera.

3.3.2.3 pixel_height

```
int CalibrationData::pixel_height = 1200
```

Pixel height of the calibrated camera.

3.3.2.4 pixel_width

```
int CalibrationData::pixel_width = 1920
```

Pixel width of the calibrated camera.

3.3.2.5 R_inv

```
Eigen::MatrixXd CalibrationData::R_inv
```

The inverse of the rotation matrix of the camera.

3.3.2.6 t

```
Eigen::MatrixXd CalibrationData::t
```

The translation vector of the camera.

The documentation for this struct was generated from the following file:

- include/Calibration.hpp

3.4 Chessboard Struct Reference

Represents a chessboard used for camera calibration.

```
#include <Calibration.hpp>
```

Public Attributes

- int start_pixel [2]

  x, y pixel coordinates of upper left corner of chessboard_region
- int pixel_width

  pixel width of the whole chessboard
- int pixel_height

  pixel height of the whole chessboard
- int world_coordinates_mm [3]

  x, y and z coordinates of upper left corner (= first element of std::vector<> corners)
- int horizontal_squares = 8

  Number of squares on the chessboard, in the horizontal direction.
- int vertical_squares = 4

  Number of squares on the chessboard, in the vertical direction.
- int square_size_mm = 10

  Real world size of each square, in millimeters.
- std::vector< cv::Point2f > corners

  Vector containing all the world coordinates of the Chessboard.

3.4.1 Detailed Description

Represents a chessboard used for camera calibration.

This structure contains information about the chessboard pattern, including its pixel coordinates, dimensions, and real-world location. It is used in the camera calibration process to map real-world coordinates to pixel coordinates and vice versa.

Author

Anton Haes

3.4.2 Member Data Documentation

3.4.2.1 corners

```
std::vector<cv::Point2f> Chessboard::corners
```

Vector containing all the world coordinates of the Chessboard.

3.4.2.2 horizontal_squares

```
int Chessboard::horizontal_squares = 8
```

Number of squares on the chessboard, in the horizontal direction.

3.4.2.3 pixel_height

```
int Chessboard::pixel_height
```

pixel height of the whole chessboard

3.4.2.4 pixel_width

```
int Chessboard::pixel_width
```

pixel width of the whole chessboard

3.4.2.5 square_size_mm

```
int Chessboard::square_size_mm = 10
```

Real world size of each square, in millimeters.

3.4.2.6 start_pixel

```
int Chessboard::start_pixel[2]
```

x, y pixel coordinates of upper left corner of chessboard_region

3.4.2.7 vertical_squares

```
int Chessboard::vertical_squares = 4
```

Number of squares on the chessboard, in the vertical direction.

3.4.2.8 world_coordinates_mm

```
int Chessboard::world_coordinates_mm[3]
```

x, y and z coordinates of upper left corner (= first element of std::vector<> corners)

The documentation for this struct was generated from the following file:

- include/Calibration.hpp

3.5 Window Class Reference

A class to encapsulate an OpenCV window for displaying frames.

```
#include <UI.hpp>
```

Public Member Functions

- Window (const std::string &name)
  Constructor that initializes the OpenCV window with the given name.
- ∼Window ()
  Destructor that destroys the OpenCV window.
- int loadFrame (cv::Mat frame)
  Loads and displays a frame in the OpenCV window.
- int loadFrame (cv::Mat ∗frame, int width, int height)
  Loads, resizes, and displays a frame in the OpenCV window.
- int load2Frames (cv::Mat ∗frame_left, cv::Mat ∗frame_right, int width, int height)
  Loads, resizes, and displays a 2 frames next to each other in the OpenCV window.

3.5.1 Detailed Description

A class to encapsulate an OpenCV window for displaying frames.

Author

Anton Haes

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Window()

```
Window::Window (
            const std::string & name)  [inline]
```

Constructor that initializes the OpenCV window with the given name.

Author

Anton Haes

Parameters

name	The name of the window.

3.5.2.2 ∼Window()

```
Window::~Window ()  [inline]
```

Destructor that destroys the OpenCV window.

Author

Anton Haes

3.5.3 Member Function Documentation

3.5.3.1 load2Frames()

```
int Window::load2Frames (
            cv::Mat * frame_left,
            cv::Mat * frame_right,
            int width,
            int height)  [inline]
```

Loads, resizes, and displays a 2 frames next to each other in the OpenCV window.

Author

Anton Haes

Parameters

frame_left	The first frame to display.
frame_right	The second frame to display
width	The width to resize both frames to.
height	The height to resize both frames to.

Returns

Return the ASCII code of the key that was pressed, or -1 if no key was pressed.

3.5.3.2 loadFrame() [1/2]

```
int Window::loadFrame (
            cv::Mat * frame,
            int width,
            int height)  [inline]
```

Loads, resizes, and displays a frame in the OpenCV window.

Author

Anton Haes

Parameters

frame	The frame to display.
width	The width to resize the frame to.
height	The height to resize the frame to.

3.5.3.3 loadFrame() [2/2]

```
int Window::loadFrame (
            cv::Mat frame)  [inline]
```

Loads and displays a frame in the OpenCV window.

Author

Anton Haes

Parameters

frame	The frame to display.

The documentation for this class was generated from the following file:

- include/UI.hpp

Chapter 4

File Documentation

4.1 include/BaslerCamera.hpp File Reference

```
#include <pylon/PylonIncludes.h>
#include <opencv2/opencv.hpp>
```

Classes

- class BaslerCamera

  A class to interface with a Basler camera or emulate a camera using a video file.

4.2 BaslerCamera.hpp

Go to the documentation of this file.

```
#ifndef BASLER_CAMERA_HPP
#define BASLER_CAMERA_HPP

#include <pylon/PylonIncludes.h>
#include <opencv2/opencv.hpp>

class BaslerCamera {
public:
    // timeout when grabbing frames from the camera
    int camera_timeout = 5000;

    BaslerCamera(float exposure_time) : is_emulated(false) {
        // Initialize Basler camera using Pylon API
        Pylon::PylonInitialize();
        camera = new Pylon::CInstantCamera(Pylon::CTlFactory::GetInstance().CreateFirstDevice());
        // sets up free-running continuous acquisition.
        camera->StartGrabbing(Pylon::GrabStrategy_LatestImageOnly);
        // configure exposure time
        GenApi::INodeMap& nodemap = camera->GetNodeMap();
        Pylon::CFloatParameter(nodemap, "ExposureTime").SetValue(exposure_time);
        // Specify the output pixel format.
        format_converter.OutputPixelFormat = Pylon::PixelType_BGR8packed;
    }

    BaslerCamera(const std::string path_to_video_file) : is_emulated(true) {
        // Use OpenCV library to read and playback video file
        if (!video_capture.open(path_to_video_file)) {
            throw std::runtime_error("Error opening video file.");
        }
    }

    ~BaslerCamera() {
```

```
        if (!is_emulated) {
            camera->Close();
            delete camera;
            Pylon::PylonTerminate();
        }
    }

    cv::Mat getFrame() {
        // Case where the camera is being emulated
        if (is_emulated) {
            cv::Mat frame;
            // Read next frame from video
            bool successful = video_capture.read(frame);
            if (!successful) {
                return cv::Mat(); // Return an empty Mat to indacte the end of the video file
            }
            return frame;
        }

        // Case where the camera is not being emulated
        if (!camera->IsGrabbing()) {
            throw std::runtime_error("Camera is currently not grabbing frames.");
        }
        // Wait for an image and then retrieve it
        camera->RetrieveResult(camera_timeout, ptr_grab_result, Pylon::TimeoutHandling_ThrowException);
        // Check if the image was grabbed successfully
        if (!ptr_grab_result->GrabSucceeded()) {
            throw std::runtime_error("Image not grabbed successfully from camera.");
        }
        // Convert image to pylonImage
        format_converter.Convert(pylon_image, ptr_grab_result);
        // Convert image to OpenCV Mat
        cv::Mat frame(ptr_grab_result->GetHeight(), ptr_grab_result->GetWidth(), CV_8UC3, (uint8_t*)pylon_image.GetBuffer());
        // Copy the frame to ensure memory safety
        frame = frame.clone();
        return frame;
    }

private:
    bool is_emulated; // variable to indicate if the camera is being emulated with a video file

    // Pylon objects for Basler camera
    Pylon::CInstantCamera* camera;
    Pylon::CGrabResultPtr ptr_grab_result;
    Pylon::CPylonImage pylon_image;
    Pylon::CImageFormatConverter format_converter;

    // OpenCV object to read video file
    cv::VideoCapture video_capture;
};

#endif // BASLER_CAMERA_HPP
```

4.3 include/Calibration.hpp File Reference

```
#include <opencv2/opencv.hpp>
#include <fstream>
#include <Dense>
#include "BaslerCamera.hpp"
#include "UI.hpp"
```

Classes

- struct Chessboard

  Represents a chessboard used for camera calibration.
- struct CalibrationData

  Contains data related to camera calibration.
- class Calibration

  Performs camera calibration using a series of chessboard patterns.

4.4 Calibration.hpp

Go to the documentation of this file.

```
#ifndef CALIBRATION_HPP
#define CALIBRATION_HPP

#include <opencv2/opencv.hpp>
#include <fstream> // library to write and read from files
#include <Dense> // Eigen library for matrices

#include "BaslerCamera.hpp"
#include "UI.hpp"

struct Chessboard {
    int start_pixel[2];
    int pixel_width;
    int pixel_height;
    int world_coordinates_mm[3];
    int horizontal_squares = 8;
    int vertical_squares = 4;
    int square_size_mm = 10;
    std::vector<cv::Point2f> corners;
};

struct CalibrationData {
    Eigen::MatrixXd P = Eigen::MatrixXd::Zero(3, 4);
    Eigen::MatrixXd K_inv;
    Eigen::MatrixXd R_inv;
    Eigen::MatrixXd t;
    int pixel_width = 1920;
    int pixel_height = 1200;
};

class Calibration {
public:

    Calibration(BaslerCamera* cam) : camera(cam) {}

    // Destructor for the Vision3D class
    ~Calibration() {}

    void findChessboardCorners(Chessboard *chessboards[]) {
        Window window("Calibration");

        for (int i = 0; chessboards[i] != NULL; i++) {
            bool found_chessboard = false;
            int start_x = chessboards[i]->start_pixel[0];
            int start_y = chessboards[i]->start_pixel[1];
            int width = chessboards[i]->pixel_width;
            int height = chessboards[i]->pixel_height;
            cv::Rect chessboard_region(start_x, start_y, width, height);
            cv::Size chessboard_size = cv::Size(chessboards[i]->horizontal_squares-1, chessboards[i]->vertical_squares-1);

            while (!found_chessboard) {
                cv::Mat frame = camera->getFrame();
                found_chessboard = cv::findChessboardCorners(frame(chessboard_region), chessboard_size, chessboards[i]->corners, cv::CALIB_CB_ADAPTIVE_THRESH);

                cv::rectangle(frame, chessboard_region, COLOR_GREEN, 3);
                window.loadFrame(frame, 768, 480);
            }

            // Change chessboard coordinates to full frame instead of chessboard_region
            for (size_t j = 0; j < chessboards[i]->corners.size(); j++) {
                chessboards[i]->corners[j].x += start_x;
                chessboards[i]->corners[j].y += start_y;
            }

            int index_top_left = 0;
            int index_top_right = chessboards[i]->horizontal_squares - 2;
            int index_bottom_left = (chessboards[i]->horizontal_squares-1) * (chessboards[i]->vertical_squares-2);

            // mirror chessboard so that x coordinates go from low to high
            if (chessboards[i]->corners[index_top_left].x > chessboards[i]->corners[index_top_right].x) {
                chessboards[i]->corners = mirrorXAxis(chessboards[i]);
            }
            // mirror chessboard so that y coordinates go from low to high
            if (chessboards[i]->corners[index_top_left].y > chessboards[i]->corners[index_bottom_left].y) {
                chessboards[i]->corners = mirrorYAxis(chessboards[i]);
            }
        }
```

```

    }

    void calibrate(Chessboard* chessboards[]) {
        // Count number of calibration points
        int num_points = 0;
        for (int i = 0; chessboards[i] != NULL; i++) {
            num_points += chessboards[i]->corners.size();
        }

        // Create and fill Matrix A (Section 4.2 from preparatory research training)
        Eigen::MatrixXd A(num_points*2, 12);
        for (int k = 0; chessboards[k] != NULL; k++) {
            int num_cols = chessboards[k]->horizontal_squares-1;
            int num_rows = chessboards[k]->vertical_squares-1;
            double Ly = (double)chessboards[k]->world_coordinates_mm[1];
            double Lz = (double)chessboards[k]->world_coordinates_mm[2];
            for (int i = 0; i < num_rows; i++) {
                double Lx = (double)chessboards[k]->world_coordinates_mm[0]; // FIX THIS
                for (int j = 0; j < num_cols; j++) {
                    int u = chessboards[k]->corners[i * num_cols + j].x;
                    int v = chessboards[k]->corners[i * num_cols + j].y;

                    int index = (k*chessboards[k]->corners.size()) + (i * num_cols + j);
                    A.row(2*index) « Lx, Ly, Lz, 1, 0, 0, 0, 0, -u*Lx, -u*Ly, -u*Lz, -u;
                    A.row(2*index+1) « 0, 0, 0, 0, Lx, Ly, Lz, 1, -v*Lx, -v*Ly, -v*Lz, -v;

                    Lx += (double)chessboards[k]->square_size_mm;
                }
                Ly -= (double)chessboards[k]->square_size_mm;
            }
        }

        // Find Eigenvectors and Eigenvalues (Section 4.2 from preparatory research training)
        Eigen::MatrixXd ATA = A.transpose() * A;
        Eigen::EigenSolver<Eigen::MatrixXd> es;
        Eigen::VectorXd eigenvalues(12, 1);
        Eigen::MatrixXd eigenvectors(12, 12);

        es.compute(ATA, true);
        eigenvalues = es.eigenvalues().real();
        eigenvectors = es.eigenvectors().real();

        // Find Eigenvalue and Eigenvector combination with smallest Loss value (Section 4.2 from preparatory research training)
        float min = FLT_MAX;
        uint8_t best_index = 0;
        for (uint8_t i = 0; i < 12; i++) {
            float lambda = eigenvalues.row(i).coeff(0, 0);
            Eigen::MatrixXd x_normalized = eigenvectors.col(i);
            float loss_func = (x_normalized.transpose() * ATA * x_normalized).coeff(0, 0) - lambda * ((x_normalized.transpose() * x_normalized).coeff(0, 0) - 1);
            if (loss_func < min) {
                best_index = i;
            }
        }

        // Fill matrix P
        Eigen::VectorXd p = es.eigenvectors().real().col(best_index);
        for (uint8_t i = 0; i < 3; i++) {
            for (uint8_t j = 0; j < 4; j++) {
                calibration_data.P(i, j) = p(i*4+j);
            }
        }

        // Create a new matrix P_1 by omitting the last column of P (section 3.1 from bachelor thesis)
        Eigen::MatrixXd P_1 = calibration_data.P.leftCols(calibration_data.P.cols() - 1);
        // Create a new matrix P_2 by extracting the last column of P (section 3.1 from bachelor thesis)
        Eigen::MatrixXd P_2 = calibration_data.P.col(calibration_data.P.cols() - 1);

        // RQ-Decomposition of P_1 (according to section 2 of bachelor thesis)
        Eigen::MatrixXd pAT = P_1.colwise().reverse().transpose(); // cfr. Equation 1 of bachelor thesis
        Eigen::HouseholderQR<Eigen::MatrixXd> qr(pAT); // Perform QR-Decomposition
        Eigen::MatrixXd q_tilde = qr.householderQ();
        Eigen::MatrixXd r_tilde = qr.matrixQR().triangularView<Eigen::Upper>();

        // Equation 8 from bachelor thesis
        Eigen::MatrixXd Q = q_tilde.transpose().colwise().reverse();
        Eigen::MatrixXd R = r_tilde.transpose().colwise().reverse().rowwise().reverse();

        // fill calibration_data
        calibration_data.K_inv = R.inverse();
        calibration_data.R_inv = Q.inverse();
        calibration_data.t = calibration_data.K_inv * P_2; // see Equation 9 from bachelor thesis
    }
```

```
    void save(const std::string& filename) {
        // Create file
        std::ofstream file(filename, std::ios::out | std::ios::binary);
        if (!file.is_open()) {
            throw std::runtime_error("Failed to open or create file for writing.");
        }

        // Write pixel width and height
        file.write(reinterpret_cast<const char*>(&calibration_data.pixel_width), sizeof(int));
        file.write(reinterpret_cast<const char*>(&calibration_data.pixel_height), sizeof(int));

        // Write matrix P
        int rows, cols;
        rows = calibration_data.P.rows();
        cols = calibration_data.P.cols();
        file.write(reinterpret_cast<const char*>(&rows), sizeof(int));
        file.write(reinterpret_cast<const char*>(&cols), sizeof(int));
        file.write(reinterpret_cast<const char*>(calibration_data.P.data()), rows * cols * sizeof(double));

        // Write matrix K_inv
        rows = calibration_data.K_inv.rows();
        cols = calibration_data.K_inv.cols();
        file.write(reinterpret_cast<const char*>(&rows), sizeof(int));
        file.write(reinterpret_cast<const char*>(&cols), sizeof(int));
        file.write(reinterpret_cast<const char*>(calibration_data.K_inv.data()), rows * cols * sizeof(double));

        // Write matrix R_inv
        rows = calibration_data.R_inv.rows();
        cols = calibration_data.R_inv.cols();
        file.write(reinterpret_cast<const char*>(&rows), sizeof(int));
        file.write(reinterpret_cast<const char*>(&cols), sizeof(int));
        file.write(reinterpret_cast<const char*>(calibration_data.R_inv.data()), rows * cols * sizeof(double));

        // Write matrix t
        rows = calibration_data.t.rows();
        cols = calibration_data.t.cols();
        file.write(reinterpret_cast<const char*>(&rows), sizeof(int));
        file.write(reinterpret_cast<const char*>(&cols), sizeof(int));
        file.write(reinterpret_cast<const char*>(calibration_data.t.data()), rows * cols * sizeof(double));

        file.close();
    }

    void drawChessboardCorners(cv::Mat* frame, Chessboard* chessboards[]) {
        for (int i = 0; chessboards[i] != NULL; i++) {
            for (size_t j = 0; j < chessboards[i]->corners.size(); j++) {
                cv::circle(*frame, chessboards[i]->corners[j], 5, COLOR_GREEN, -1);
            }
        }
    }



private:
    BaslerCamera* camera;
    CalibrationData calibration_data;

    std::vector<cv::Point2f> mirrorXAxis(Chessboard* chessboard) {
        std::vector<cv::Point2f> mirroredVector(chessboard->corners.size());

        int num_cols = chessboard->horizontal_squares-1;
        int num_rows = chessboard->vertical_squares-1;

        for (int i = 0; i < num_rows; i++) {
            for (int j = 0; j < num_cols; j++) {
                mirroredVector[i * num_cols + j] = chessboard->corners[i * num_cols + num_cols - 1 - j];
            }
        }

        return mirroredVector;
    }

    std::vector<cv::Point2f> mirrorYAxis(Chessboard* chessboard) {
        std::vector<cv::Point2f> mirroredVector(chessboard->corners.size());

        int num_cols = chessboard->horizontal_squares-1;
        int num_rows = chessboard->vertical_squares-1;

        for (int i = 0; i < num_rows; i++) {
            for (int j = 0; j < num_cols; j++) {
                mirroredVector[i * num_cols + j] = chessboard->corners[(num_rows - 1 - i) * num_cols +
```

```
j];
            }
        }

        return mirroredVector;
    }
};

#endif // CALIBRATION_HPP
```

4.5 include/UI.hpp File Reference

```
#include <opencv2/opencv.hpp>
```

Classes

- class Window
  A class to encapsulate an OpenCV window for displaying frames.

Macros

- #define COLOR_RED cv::Scalar(0, 0, 255)
- #define COLOR_GREEN cv::Scalar(0, 255, 0)
- #define COLOR_BLUE cv::Scalar(255, 0, 0)
- #define COLOR_YELLOW cv::Scalar(0, 255, 255)
- #define COLOR_GREY cv::Scalar(127, 127, 127)

4.5.1 Macro Definition Documentation

4.5.1.1 COLOR_BLUE

```
#define COLOR_BLUE cv::Scalar(255, 0, 0)
```

4.5.1.2 COLOR_GREEN

```
#define COLOR_GREEN cv::Scalar(0, 255, 0)
```

4.5.1.3 COLOR_GREY

```
#define COLOR_GREY cv::Scalar(127, 127, 127)
```

4.5.1.4 COLOR_RED

```
#define COLOR_RED cv::Scalar(0, 0, 255)
```

4.5.1.5 COLOR_YELLOW

```
#define COLOR_YELLOW cv::Scalar(0, 255, 255)
```

4.6 UI.hpp

Go to the documentation of this file.

```
#ifndef UI_HPP
#define UI_HPP

#include <opencv2/opencv.hpp>

#define COLOR_RED       cv::Scalar(0, 0, 255)
#define COLOR_GREEN     cv::Scalar(0, 255, 0)
#define COLOR_BLUE      cv::Scalar(255, 0, 0)
#define COLOR_YELLOW    cv::Scalar(0, 255, 255)
#define COLOR_GREY      cv::Scalar(127, 127, 127)

class Window {
public:

    Window(const std::string& name) : window_name(name) {
        cv::namedWindow(window_name, cv::WINDOW_AUTOSIZE);
    }

    ~Window() {
        cv::destroyWindow(window_name);
    }

    int loadFrame(cv::Mat frame) {
        cv::imshow(window_name, frame);
        return cv::waitKey(1);
    }

    int loadFrame(cv::Mat* frame, int width, int height) {
        cv::resize(*frame, *frame, cv::Size(width, height));
        cv::imshow(window_name, *frame);
        return cv::waitKey(1);
    }

    int load2Frames(cv::Mat* frame_left, cv::Mat* frame_right, int width, int height) {
        // resize the left frame to match the output size
        cv::resize(*frame_left, *frame_left, cv::Size(width, height));
        // resize the right frame to match the output size
        cv::resize(*frame_right, *frame_right, cv::Size(width, height));
        // create a new frame, and fill it with the left and right frame
        cv::Mat combined_frame(width*2, height, CV_8UC3);
        cv::hconcat(*frame_left, *frame_right, combined_frame);
        // show the frame
        cv::imshow(window_name, combined_frame);
        return cv::waitKey(1);
    }

private:
    std::string window_name; // The name of the OpenCV window.
};

#endif // UI_HPP

```

4.7 src/main.cpp File Reference

```
#include <opencv2/opencv.hpp>
#include "BaslerCamera.hpp"
#include "UI.hpp"
#include "Calibration.hpp"
```

Functions

- int main (int argc, char **argv)

Variables

- Chessboard lower_chessboard_tower
- Chessboard upper_chessboard_tower
- Chessboard lower_chessboard_car
- Chessboard upper_chessboard_car

4.7.1 Function Documentation

4.7.1.1 main()

```
int main (
            int argc,
            char ** argv)
```

4.7.2 Variable Documentation

4.7.2.1 lower_chessboard_car

```
Chessboard lower_chessboard_car
```

Initial value:

```
= {
    .start_pixel = {685, 500},
    .pixel_width = 580,
    .pixel_height = 160,
    .world_coordinates_mm = {-30, 20, 26}
}
```

4.7.2.2 lower_chessboard_tower

```
Chessboard lower_chessboard_tower
```

Initial value:

```
= {
    .start_pixel = {685, 675},
    .pixel_width = 580,
    .pixel_height = 275,
    .world_coordinates_mm = {-30, 20, 5}
}
```

4.7.2.3 upper_chessboard_car

```
Chessboard upper_chessboard_car
```

Initial value:

```
= {
    .start_pixel = {775, 365},
    .pixel_width = 420,
    .pixel_height = 115,
    .world_coordinates_mm = {-30, 60, 36}
}
```

4.7.2.4 upper_chessboard_tower

```
Chessboard upper_chessboard_tower
```

Initial value:

```
= {
    .start_pixel = {720, 500},
    .pixel_width = 520,
    .pixel_height = 175,
    .world_coordinates_mm = {-30, 60, 15}
}
```

Index