Project in Image Processing and Analysis (234329) - Stereo Camera

Yaniv Wolf (318173663)

June 27, 2022

Contents

1 Abstract **2**

2 Background **2**

3 My Implementation **3**
3.1 Goal 3
3.2 The Cameras 3
3.3 Streaming 4
3.4 Calibration 4
3.5 HW Triggering 5
3.6 Disparity calculation 6

4 Validation **7**
4.1 Comparison with Intel RealSense D415 7
4.2 Baseline size 7

5 Future Work **8**

References **8**

A Appendix: Installing and running the program **9**
A.1 Installations and preparation 9
A.1.1 Installing Conda 9
A.1.2 Cloning the repository and installing the conda environment 9
A.1.3 Installing Vimbapython 10
A.1.4 Installing Cuda 11
A.1.5 Installing PyTorch 12
A.1.6 Installing Raft-Stereo 12
A.1.7 Initial Run 12
A.2 Running the program 13
A.2.1 Repository structure 13
A.2.2 Parameters 15
A.2.3 Streaming 16
A.2.4 Calibration 18
A.2.5 Capturing 19
A.2.6 Saving disparity maps and point clouds 19
A.2.7 Viewing point clouds 20

1 Abstract

While traditional cameras are constrained to two dimensional images, it is possible to utilize several cameras to create systems capable of capturing three-dimensional images. Such systems are called stereo camera systems, and are capable of capturing the depth of every point in the scene, using a stereo matching algorithm and the distance between the cameras. In this project, I created a stereo camera from scratch, which includes wiring the cameras for synchronized hardware triggering, and building an interface capable of streaming and capturing depth images, generating and viewing point clouds, and interactively performing internal and stereo calibration of the camera array.

Keywords: Stereo-Alignment, Calibrated Stereo, Uncalibrated Stereo, Stereo Matching, Disparity, Rotation, Translation, Rectification, Epipolar Lines.

2 Background

Three-dimensional cameras are at the core of numerous important applications such as autonomous driving, 3D scanning, and augmented reality. The basic concept of stereo vision is inspired by the depth perception of human eyesight: given two **stereo-aligned** images (all identical points between the images share the same y coordinates), it is possible to create a depth map of the image using **calibrated stereo** - That is, for each point in one image, find the corresponding point in the second image and calculate the difference in their x coordinates (since the y coordinates are the same because of the alignment). This difference is called the **disparity**. Using a disparity map, the depth of each point can be found given the intrinsic parameters of the camera and the relative position between the cameras.

However, it is nearly impossible to stereo-align cameras physically, since that would require precision crafting a stand for them, which would be prone to physical changes. Similarly to the calibrated stereo problem, the problem of stereo imaging without alignment is called **uncalibrated stereo**. To solve this harder problem, it would be helpful to first convert it into the calibrated stereo problem. This is a process known as **rectification** and it involves several steps:

1. An intrinsic calibration of each camera on its own. During this calibration, the intrinsic matrix of the camera is discovered, as well as the distortion coefficients. The intrinsic matrix of a camera is a 3×3 matrix consisting of the focal lengths of the camera in the x and y directions (f_x and f_y respectively), as well as the zero origin coordinates in the x and y directions (c_x and c_y respectively):

$$\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

   The calibration is performed using a set of known points in the image, such as corners in a chessboard. By taking several images of the chessboard, in various positions and angles, the parameters of the camera can be extracted using a constrained Least Squares approximation: Given the coordinates in the image and the real world coordinates (which are known since the chessboard is of known size), it is possible to build a system of equations to extract a 3×4 projection matrix. The intrinsic parameters can then be extracted from this matrix using QR decomposition.

2. Once the intrinsic parameters are known, the next step is to perform the stereo calibration. This means finding the **rotation** and **translation** matrices that transform the coordinates of one camera to the coordinates of the other. This is done again using a set of known points in both images, and solving a constrained Least Squares problem to find the fundamental matrix that transforms from one coordinate system to the other. From there, the rotation and translation can be extracted using the intrinsic parameters of the cameras and SVD.

3. Once the rotation and translation matrices are known, the next step is to rectify the images, meaning project them onto a common plane, to achieve stereo-alignment. This is done by converting the rotation and translation matrices into a 3×3 rectification matrix, that will act as a linear transformation on the image. Later, the images are warped to fix the distortion. At this stage, the images are rectified.

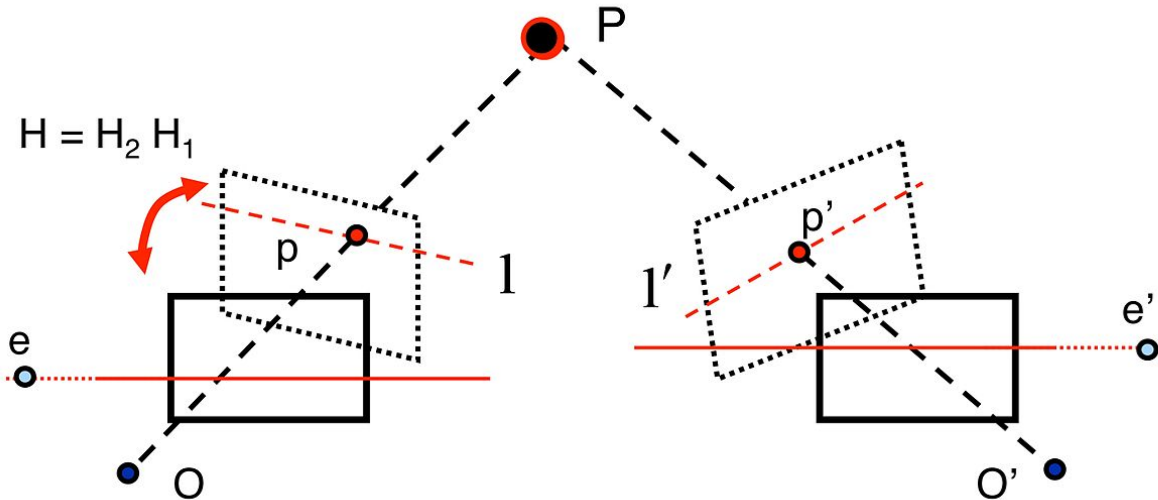


Figure 1: Visualization of the stereo rectification process [2].

As can be seen in figure 1, once the images are stereo-aligned, the problem becomes that of calibrated stereo, which was mentioned earlier. The naive solution for calibrated stereo is to segment one image into blocks, and using some similarity metric, for each block, find the corresponding block on the other image that is the most similar to the given block (this involves searching a single line (called **epipolar line**) rather than the whole image due to the rectification). The current state-of-the art solutions work using neural networks, and in this project we will use the RAFT-Stereo model [1][4] as the stereo matching algorithm.

3 My Implementation

3.1 Goal

The goal of this project was to create a 3D stereo camera array and interface "from scratch", which will be used to test new stereo algorithms later on. This includes building an interface that can:

- Stream live **synchronized** and **rectified** frames from all the cameras.
- Control the exposure for different lighting conditions.
- Perform intrinsic and stereo calibration using varying sizes of chessboards.
- Stream the resulting disparity images after calibration in real-time.
- Capture raw and rectified images, and create and view disparity maps and point clouds.

3.2 The Cameras

The camera sensors were from Allied Vision, model Alvium U-507c [3]. These cameras were fitted with $2.5mm$ lens, and they were mounted on a 3D-printed rig, capable of rotating in the x and y directions. The distance between two adjacent cameras was $20cm$. There are two cables connected to each camera: One is the USB cable that connects the cameras to a computer for frame acquisition, and the second is a serial cable used for the HW trigger mechanism to ensure that all cameras trigger at the exact same time.

Figure 2: The camera array.

3.3 Streaming

The first challenge was to understand how to use these new cameras, which were never before used in the lab. I downloaded the official Python library for these cameras, called VimbaPython [5]. I based my frame acquisition on an example for multithreaded frame acquisition from multiple cameras at the same time. The general idea is to have a main thread which spawns a producer thread for each camera and a single consumer thread that displays all the acquired frames from a shared queue. The cameras work in such a way that there is no configuration file saved onboard, but instead the settings are loaded each time the camera connects. There are several options for triggering the cameras, and at first I used the default option which was just sending a signal from software and letting the cameras take frames freely and queue them. This turned out to be a bad idea, which will be discussed later on in section 3.5.

3.4 Calibration

After achieving a steady stream from the cameras, the next step was to perform calibration. As mentioned earlier, there are two types of calibrations needed to create a stereo camera:

- Intrinsic calibration, individual per camera. This calibration is used to determine the intrinsic parameters of each camera, which compose the intrinsic matrix. It also computes the distortion coefficients.
- 3D stereo calibration, which is performed on all cameras simultaneously. This calibration is used to determine the location of one camera relative to another camera, and compute the rectification matrix. Using this matrix, it is possible to rectify images and project all of the frames to a common plane. This ensures that the horizontal lines in each image contain the same parts of the scene, thus reducing the stereo matching to a linear search rather than searching the entire image.

In order to calibrate the cameras, a chessboard is used.

For the first stage of calibration, 15 images are taken from each camera, while moving and tilting either the chessboard or the cameras, to achieve a wide variety of angles. While experimenting, I discovered that fewer images lead to a bad calibration since there wasn't enough data, and too many images won't necessarily improve the calibration, and only cost more resources and time (also, it increases the chance of a having a bad image in the set of images which can damage the entire calibration). Once the images are obtained, the program extracts the corner locations using the OpenCV *findChessboardCorners()* method, and given prior knowledge of the dimensions of the chessboard, it calculates the intrinsic matrix and the distortion coefficients using the OpenCV *calibrateCamera()* method.

For the second stage of calibration, another 15 images are taken from all cameras simultaneously, and then the stereo calibration is performed, using the intrinsic matrices and distortions calculated in the first stage, with the OpenCV *stereoCalibrate()* method.

One important note, is that the program will only allow to take an image for calibration when all relevant cameras have visible chessboards, all corners are detected and the reprojection error is low (the reprojection

error is calculated by taking the MSE between the set of points that were detected, and a projection of a perfect grid to the image using the calculated intrinsic matrix with the OpenCV *projectPoints()* method). To allow for more real-time performance, the OpenCV *checkChessboard()* method which is quicker is first used to check if a chessboard exists in the frame before searching for points. These steps ensure as much as possible that the reprojection error at the end of the calibration will also be low enough.

3.5 HW Triggering

However, after many attempts, the results of the stereo calibration were still poor and the resulting rectified images were mostly blank, since the translation vector was large. Further investigation and experimenting lead to the conclusion that the cause is the fact that the cameras were sending frames in an uncontrolled manner, leading to differences of up to 100 milliseconds between images from different cameras (This was checked using a screen with a running stopwatch, and comparing the time shown on the screen from each camera after a capture was taken).

The first solution I tried was a software trigger mechanism that sent a single frame from each camera to the computer every time it was called, instead of sending frames uncontrollably. This would solve the problem of uncontrolled frames from the cameras, however since it was dependant on the fact that all signals arrive at the same time to all cameras, this did not work out well (I even noticed that cameras which were connected to adjacent USB ports on my computer were closer in synchronization than those connected on opposite sides).

Therefore, I had to utilize the HW trigger mechanism mentioned earlier. This was done by using the Line0 port of the camera, and wiring all of the cables together to one pin, which could send a signal to all cameras at the same time. This pin was connected to an Arduino Nano board, which was programmed to send a short rising-edge signal once it receives a signal from the computer in control. This means that through one signal from the computer (rather than multiple signals), all of the cameras will receive the signal at the same time and send a frame. The program will then process those frames, and only then a new signal will be sent. This ensures that no frames are discarded, and indeed the cameras were now synchronized.

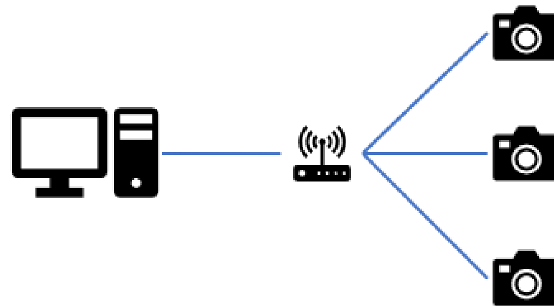


Figure 3: Schematic drawing of the HW triggering mechanism.

When performing the same check with the stopwatch, the cameras all triggered at the same time:

Figure 4: Visualization of the HW triggering mechanism - All three cameras trigger at the same time.

3.6 Disparity calculation

After the cameras have been calibrated and the frames were rectified, it was possible to start the creation of stereo images. For this project, the RAFT-Stereo network [1][4], trained on the Middlebury dataset [6] was used, with a slight modification to improve stability: Since we are dealing with a stream of frames, and not individual unrelated frames, we can use the previous disparity map as an initial guess for the current one during inference. I edited the RAFT-Stereo code to allow that. When streaming disparity images in real-time, it can be seen that after a few seconds, if the scene is static, the disparity map becomes more stable and doesn't change as much, a feature that would be less likely to happen with an initial guess of zero at every frame.

The RAFT-Stereo network is composed of several stages:

- A CNN feature extractor, that passes the features to a correlation pyramid, which calculates a 3D correlation matrix between pixels of the same height (hence the importance of rectification).
- A context encoder which extracts additional features and an initial hidden state for the RNN
- An iterative GRU RNN for disparity calculation and refinement. The number of iterations in the RNN directly affects the runtime of the network, but also directly affects the quality of the resulting disparity map. For real-time viewing of the disparity map inside the program, I used a faster implementation with 8 iterations, and for saving the disparity maps post-recording, I used a slower but more refined implementation with 32 iterations. Notice, that since I am passing the previous disparity map as the initial guess while calculating the next disparity map while streaming, then if the scene is static, the number of iterations accumulates between frames, and therefore after several static frames the quality of the depth map will be better.

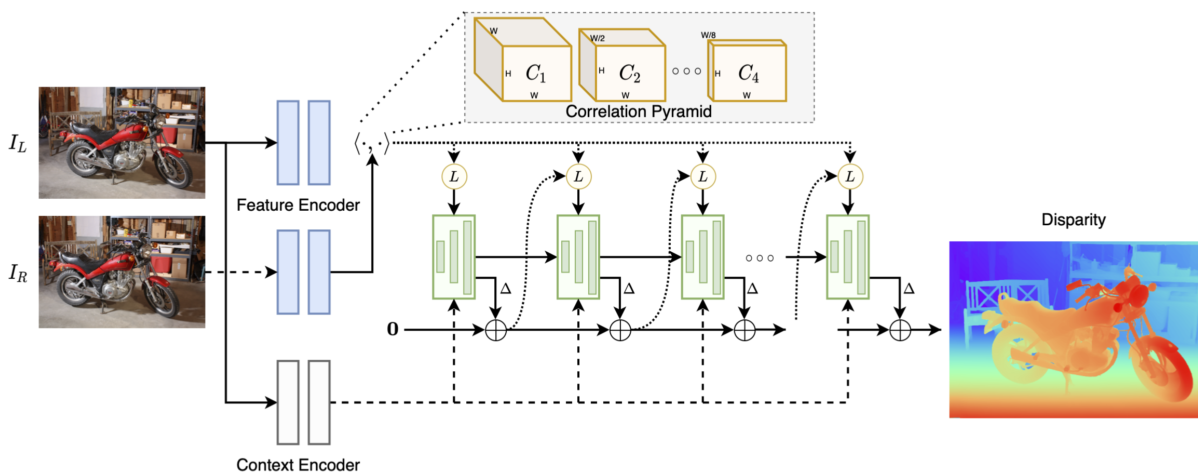


Figure 5: The RAFT-Stereo network [1]

This network was shown to improve the benchmark results on the Middlebury dataset [1].

After calculating the disparity map, a point cloud can be created using the OpenCV $reprojectTo3D()$ method. Details about how to save and view pointclouds, including cropping and outlier removal, can be found in appendix A.2.6 and appendix A.2.7.

4 Validation

4.1 Comparison with Intel RealSense D415

Below are point clouds of the same scene, taken by the Intel RealSense D415 camera and my camera:

Figure 6: Left: Point cloud from my camera, with outlier removal. Right: Point cloud from Intel RealSense D435 camera.

As can be seen, while the Intel camera has faster real-time capabilities, the resulting point cloud is noisy and has many holes, compared to the point cloud created by my camera.

4.2 Baseline size

The baseline, or the distance between the two cameras, can affect the quality of the resulting stereo image. There is a trade-off between the amount of common space between the two images (more common space works better for closer objects), and the ability to calculate depth for objects farther away (where the more common space, the worse it'll be since it's equivalent to the cameras being at the same spot). Since we have three cameras, we can try creating the same disparity image from different pairs of cameras:

Figure 7: Left: Point cloud from 20cm baseline, with outlier removal. Right: Point cloud from 40cm baseline, with outlier removal.

As can be seen, for closer objects, a smaller baseline gives better results.

5 Future Work

This project creates an infrastructure for testing new stereo algorithms. It is built in a modular way, such that for example, the algorithm that calculates the disparity map can be swapped with ease. It also includes three cameras instead of two. This opens up new possibilities:

1. Use three cameras instead of two for the stereo image calculation - The larger baseline will be better for detecting depth of farther away objects, and the smaller baseline will be better for closer objects. Building a model that merges information between different pairs of images can improve the overall quality of the resulting stereo image.

2. Improve real-time capabilities of the model, for faster streaming of stereo images.

3. Improve depth calculation of texture-less flat surfaces such as blank walls.

References

[1] Lahav Lipson, Zachary Teed, and Jia Deng. "RAFT-Stereo: Multilevel Recurrent Field Transforms for Stereo Matching". In: *arXiv:2109.07547 [cs]* (Sept. 2021). arXiv: 2109.07547. URL: `http://arxiv.org/abs/2109.07547` (visited on 06/02/2022).

[2] *Image rectification*. en. Page Version ID: 1083286715. Apr. 2022. URL: `https://en.wikipedia.org/w/index.php?title=Image_rectification&oldid=1083286715` (visited on 06/02/2022).

[3] *AlliedVision Alvium1800 U-507c Datasheet*. URL: `https://www.alliedvision.com/fileadmin/pdf/en/Alvium_1800_U-507c_Closed-Housing_C-Mount_Standard_DataSheet_V1.6.0_en.pdf`.

[4] *RAFT-Stereo Github Repository*. URL: `https://github.com/princeton-vl/RAFT-Stereo`.

[5] *VimbaPython Github Repository*. URL: `https://github.com/alliedvision/VimbaPython`.

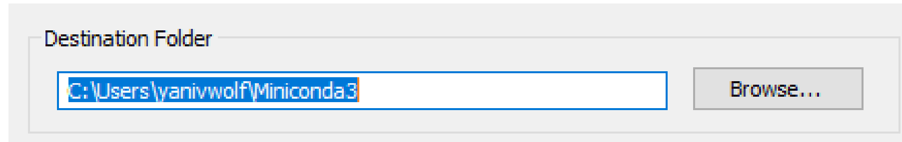
[6] *vision.middlebury.edu/stereo/data*. URL: `https://vision.middlebury.edu/stereo/data/` (visited on 06/02/2022).

A Appendix: Installing and running the program

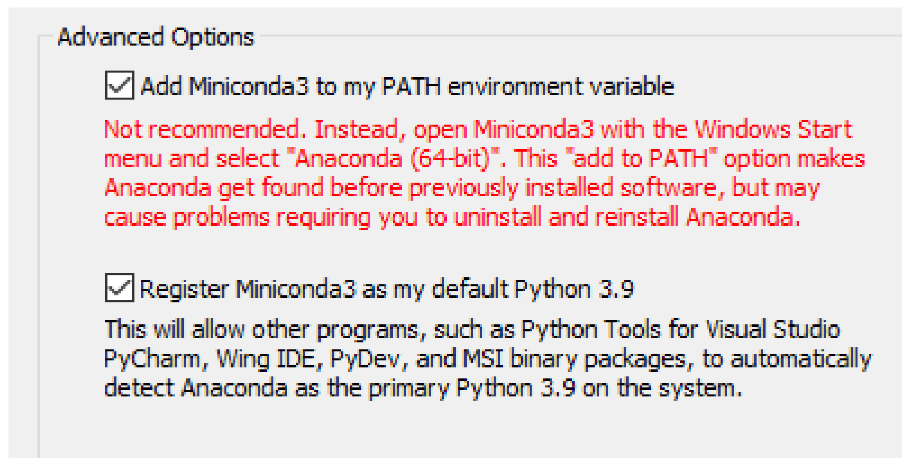
A.1 Installations and preparation

A.1.1 Installing Conda

Go to `https://docs.conda.io/en/latest/miniconda.html` and install the appropriate Miniconda3 installation. Don't change the default destination folder, and take note of it:



In the advanced options, add the Miniconda3 PATH variable:



A.1.2 Cloning the repository and installing the conda environment

Go to the destination folder for the repository, and clone the repository into that folder. Open a command prompt at the root of the repository, and run:

```
conda env create -f environment.yml
```

This should take several minutes. If there are any conflicts, solve them by editing the environment.yml file according to the message, then running

```
conda env remove -n stereo
```

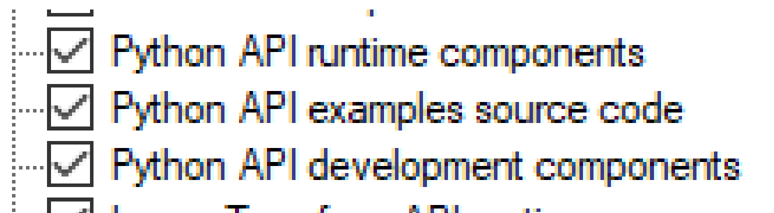
and then re-running

```
conda env create -f environment.yml
```

A.1.3 Installing Vimbapython

Go to `https://www.alliedvision.com/en/products/vimba-sdk/#c1497` and install the Windows installation. Open the installer:

Take note of the target folder. Next click *Custom Selection*, and make sure that ALL check-boxes are selected, including these:



Click *OK* and then *Start*. Once the installation is finished, make sure the following is marked:

Press *Exit* and follow the installation directions.

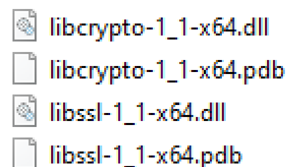
Once that is finished, go to the previously noted target folder, and enter the VimbaPython sub-folder. Enter the *Source* sub-folder, and open a command prompt there. Activate the conda environment with:

```
conda activate stereo
```

Then, run the command

```
python -m pip install .
```

If there is an error regarding something with SSL, this can be solved by going to the Miniconda3 folder (which was previously noted in step 1), and copying the following files from $Miniconda3/Library/bin$ to $Miniconda3/DLLs$:



A.1.4 Installing Cuda

Go to `https://developer.nvidia.com/cuda-10.2-download-archive?target_os=Windows&target_arch=x86_64&target_version=10&target_type=exenetwork` And install the base installer and patches. Follow the instructions within each installation.

A.1.5 Installing PyTorch

Copy the following command to a command prompt with the conda environment activated:

```
pip install torch==1.9.0+cu102 torchaudio torchvision -f https://download.pytorch.org/whl/
    torch_stable.html
```

A.1.6 Installing Raft-Stereo

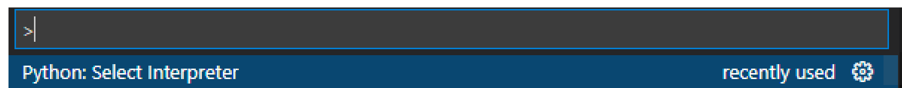
Inside the repository, go to *Stereo_Camera_Project/code/RAFT_Stereo_main/sampler*, open a command prompt inside the folder with the conda environment activated and run the following command:

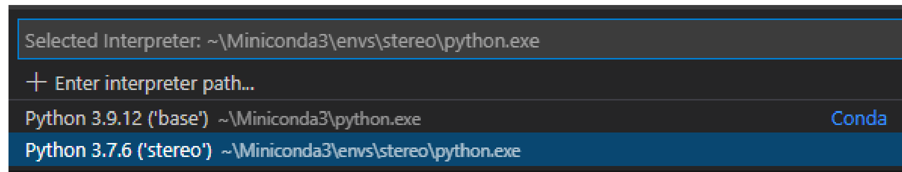
```
python setup.py install
```

This should take a while to finish and output a lot of logs while installing. This step is slightly problematic and requires Cuda 10.2, otherwise it won't work.

A.1.7 Initial Run

Open the root of the repository with VSCode, and select the python interpreter By typing $Ctrl + p$:





Open *Stereo_Camera_Project/code/stereo/stereo_stream.py*, and run it using $Ctrl + f5$.
If there are errors, open a command prompt with the environment activated and run the following commands line-by-line:

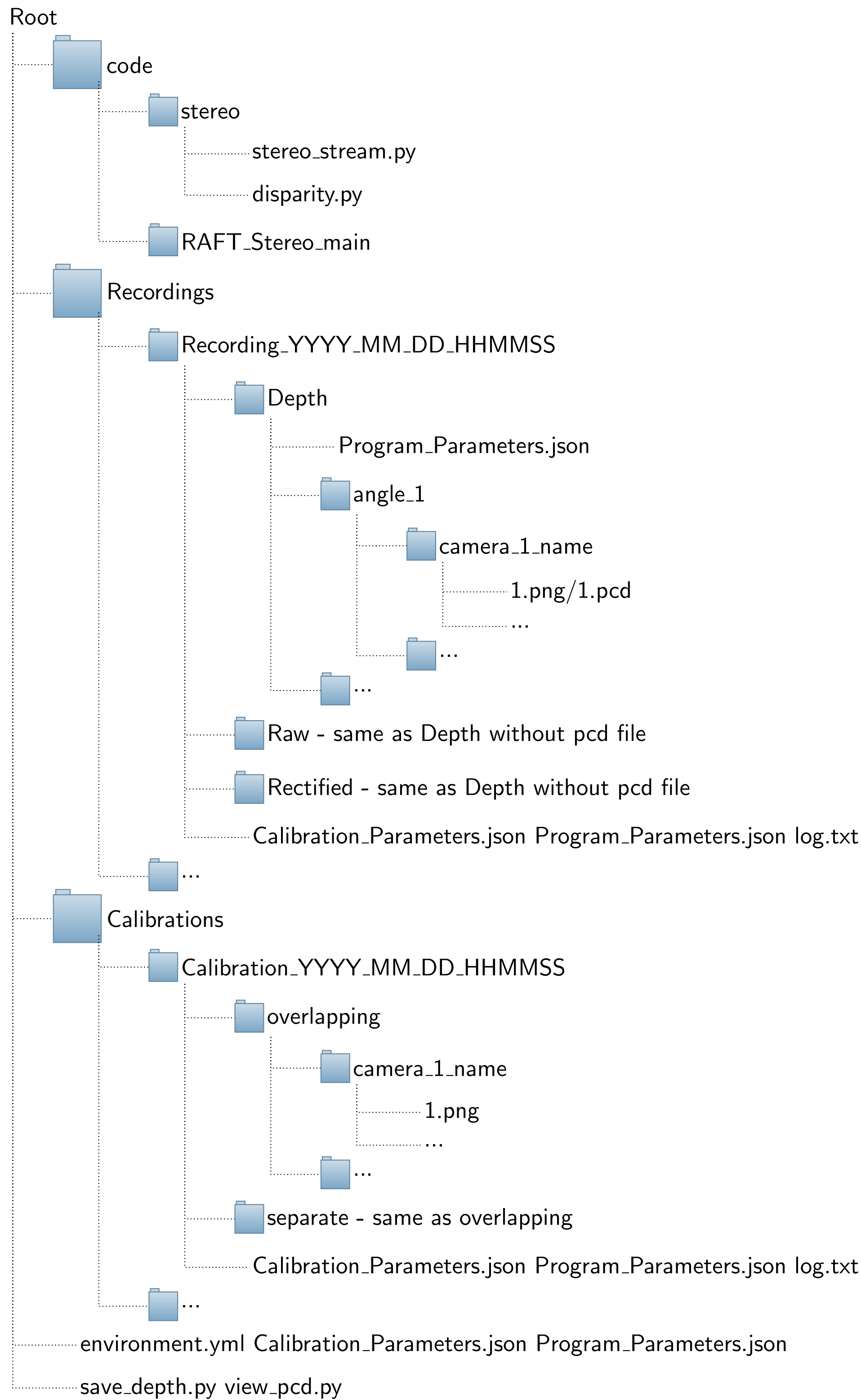
```
pip install -U numpy
pip install -U Pillow
pip install -U matplotlib
pip uninstall scipy
pip install scipy
```

Afterwards, try running again.

A.2 Running the program

A.2.1 Repository structure

The repository contains the following files:



In the root directory:

- *environment.yml*: The environment file for Conda.
- *Calibration_Parameters.json*: The latest calibration results. This file will be used to get the calibration parameters when running the program.
- *Program_Parameters.json*: The program parameters, to be changed by the user. See section A.2.2.
- *save_depth.py*: Script to save disparity maps and point clouds from existing recordings. See section A.2.6.
- *view_pcd.py*: Script to view a point cloud file in 3D. See section A.2.7.

There are also three sub-directories:

1. *Recordings*: Contains a folder for each new recording. The folder names are in the following format: *"Recording_YYYY_MM_DD_HHMMSS"*. Each folder has three sub-directories:
   - *Depth*, containing disparity images and point clouds.
   - *Raw*, containing raw images before rectification
   - *Rectified*, containing rectified images.

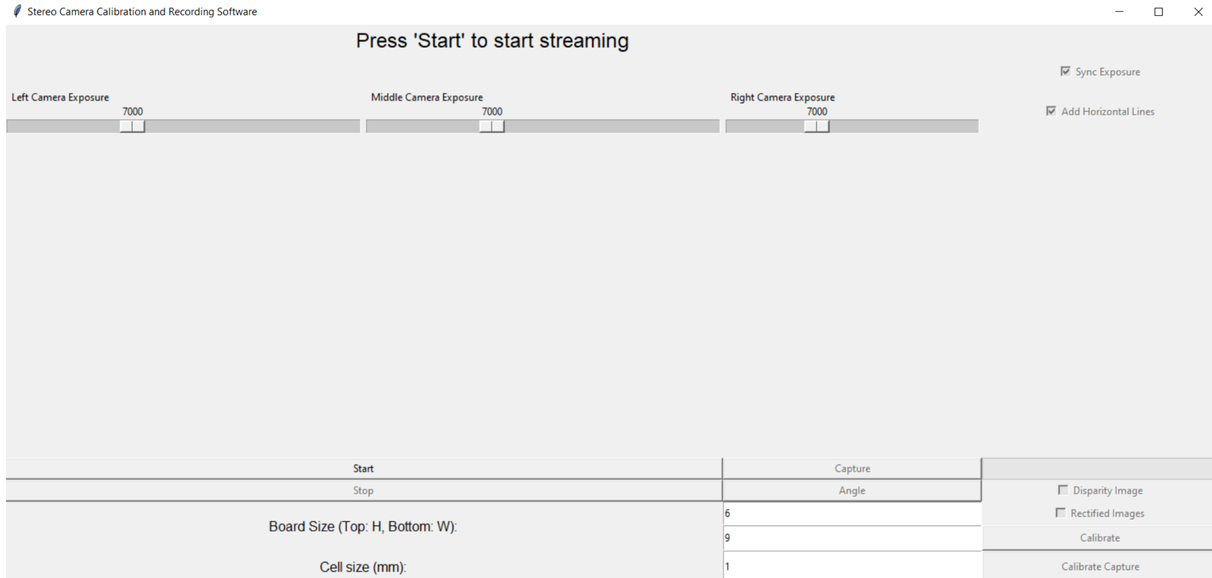
   The images are saved within each sub-directory, such that each camera has its own folder. In addition, the calibration parameters, program parameters and log of the specific run are saved in the recording folder as well. The program parameters file used when saving point clouds is also saved inside the *Depth* directory.
2. *Calibrations*: Contains the images used in a speccific calibration. There are sub-directories for the first part of the calibration (separate) and the second part (overlapping). The calibration and program parameters, as well as the log, are also saved.
3. *code*: Contains the code for the program. It contains two folders:
   - *stereo*: The folder containing the code for most of the interface: *stereo_stream.py* contains most of the code, and *disparity.py* contains the code for computing the disparity image given two images.
   - *RAFT_Stereo_main*: The folder containing the library of the network used to calculate the disparity.

A.2.2 Parameters

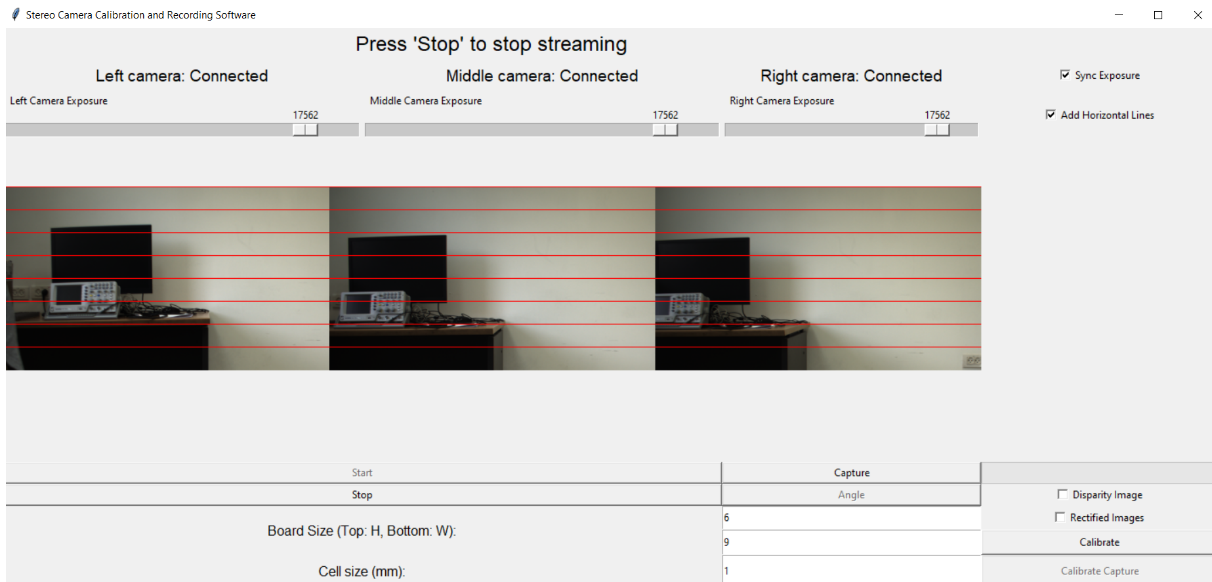
- *LEFT_CAMERA_ID*: The ID of the left camera.
- *MIDDLE_CAMERA_ID*: The ID of the middle camera.
- *RIGHT_CAMERA_ID*: The ID of the right camera.
- *MAIN_CAMERA_ID*: The ID of the camera that will be the reference point in the calibration process.
- *ID_TO_NAME*: A mapping between camera ID and the camera name.
- *LEFT_STEREO*: The ID of the camera that will be the left camera in the stereo algorithm. Usually the same as the main camera.
- *RIGHT_STEREO*: The ID of the camera that will be the right camera in the stereo algorithm.
- *FRAME_WIDTH*: The width of the frame (maximal size these cameras support is 2464).
- *FRAME_HEIGHT*: The height of the frame (maximal size these cameras support is 1944).
- *INITIAL_DOWNSIZING_FACTOR*: The initial percentage to downsize the images being displayed, to better fit the screen (doesn't affect the size of saved images).
- *RECTIFICATION_CROP_PERCENTAGE*: The percentage of the image to crop after rectification, to avoid blank parts in the edges of the image.
- *GPU_SERVER_IP*: The IP address of the GPU server. If the local machine has a GPU, write 'local' as the IP address.
- *FAST_LOCAL_STREAM*: Set to true for more real-time disparity calculation. Lower quality disparity.
- *USB_THROUGHPUT*: Might differ between computers, but it's super critical because otherwise the stream might get stuck. Default is 150000000, might need to be 50000000 in some computers.

A.2.3 Streaming

When running the program, the main screen opens:



First, start the GUI using the *Start* button:



Once all camera streams are up, adjust the exposure to an adequate level, and position the camera array in front of the scene being captured. There are several important check-boxes:

- *Sync Exposure*: Syncs the exposure between all cameras. When this is marked, moving any exposure slider moves all exposure sliders together. It is recommended to always leave this marked.
- *Add Lines*: Adds horizontal lines to all images to check if the rectification worked well.

- *Rectified Images*: Shows the rectified images instead of the raw images. Note that the cameras need to be calibrated for this to work, see section A.2.4.
- *Disparity Images*: Shows two streams: The main camera rectified stream, and the disparity image stream:

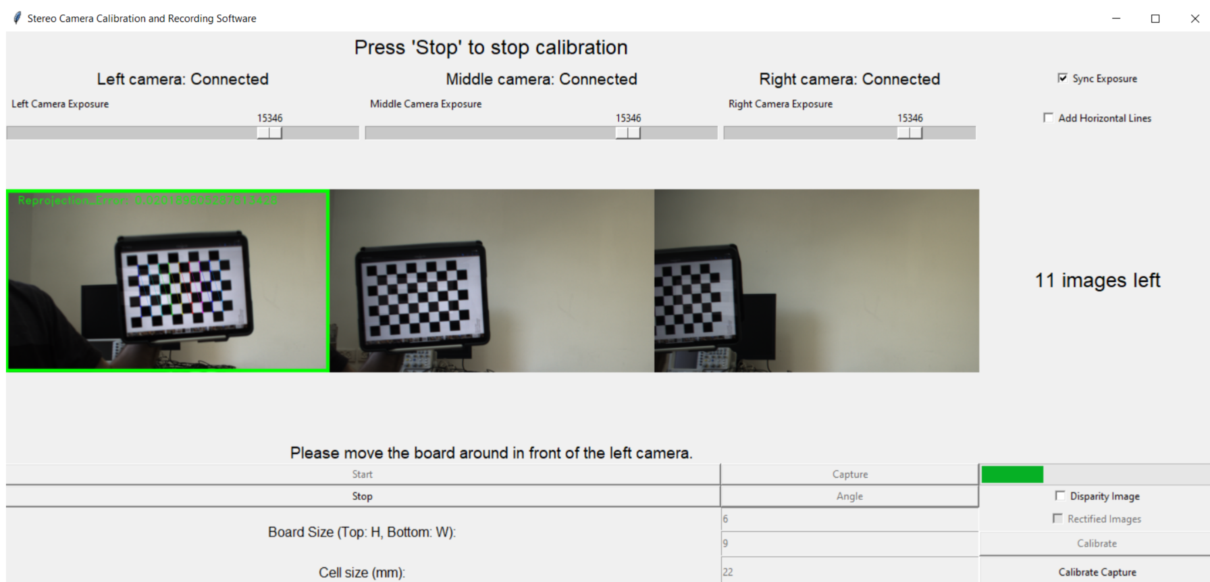
Important note: The stream must always be stopped using the *Stop* button before closing the program, otherwise all cameras have to be unplugged and plugged again.

A.2.4 Calibration

On the first time the program is opened, or whenever a *Calibration_Parameters.json* file is missing or partial, the program will ask to perform a calibration. This involves several steps:

1. First, open the GUI, press *Start*, and make sure all cameras are properly connected, the exposure is synced and lighting conditions are adequate (not too bright but not too dark. This can be easily adjusted during the first calibration). Make sure the chessboard is visible and in focus in all relevant cameras (currently being calibrated, marked by a red/green border), and takes up most of the frame. Enter the chessboard dimensions in the GUI, before pressing *Calibrate*. Note that the dimensions are according to how many inner points exist in the chessboard, and not how many squares there are in each direction. The cell size is in mm.

2. Once everything is ready, click *Calibrate* to start the calibration. At first, the internal calibration is performed, and a red/green border will appear around the camera/s being calibrated:



   The border is red when the chessboard isn't visible, or when the reprojection error is too high. On the right-hand side of the screen, there is a progress bar with the remaining amount of images to take. On the bottom right, there is a button for capturing images for calibration. Note that the button will only work when all relevant borders of cameras being calibrated are green. Move the camera/chessboard, such that a variety of angles and locations of the chessboard are achieved, and that the chessboard has appeared in most of the frame. Repeat this process for each camera.

3. Once the internal calibration is performed, the stereo calibration is performed. This involves a similar process to the previous step, except this time the images are taken from all of the cameras at the same time, meaning that all of the cameras need to have a green border around them for the capture to work.

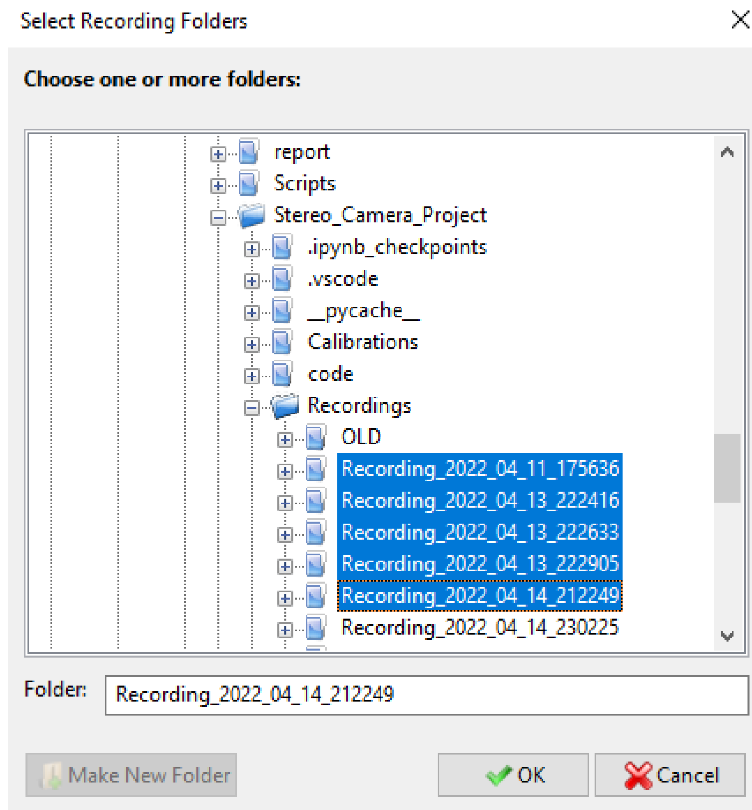
After performing the stereo calibration, a new *Calibration_Parameters.json* file will be saved in the root directory, as well as a new calibration directory in the *Calibrations* folder. There will be a message saying that the calibration was completed successfully. At this stage the GUI can be safely closed and reopened for stereo imaging use.

A.2.5 Capturing

Start streaming as before. By default, all captures are saved in a folder called *angle_1*. Pressing the *Capture* button will save a new capture. Pressing the *Angle* button will add a new angle folder to the recording and start saving images there. Note that an angle folder has to contain at least one image before creating a new angle folder.

A.2.6 Saving disparity maps and point clouds

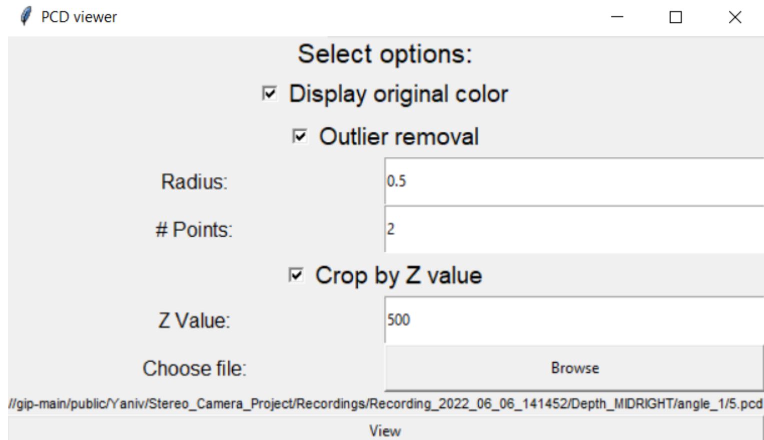
Disparity maps and point clouds are saved post-recording via a script called *save_depth.py* in the root of the repository. The script is to be opened, and the relevant folders of recordings can be selected using the GUI, as follows:



Select *OK* and wait patiently.

A.2.7 Viewing point clouds

Point clouds can be viewed using the *view_pcd.py* script in the root directory. Running the script, the following windows opens up:



The following settings are available:

- *Display original color*: If checked, displays the pointcloud with the color from the original RGB image, otherwise displays with a heatmap.
- *Outlier removal*: Removes outliers from the point cloud. If checked, the radius and the number of points inside the radius must be specified.
- *Crop by Z value*: Crops the pointcloud based on the Z value of the point (useful for taking a 3D image of a specific object for example). If checked, the Z value must be specified.
- *Browse*: Browse for the desired *.pcd* file. The filename will appear in the following row.
- *View*: View the pointcloud.

Once the file and settings are chosen, click *view* and the point cloud will open in 3D. pressing *Shift+Left click* on a point in the point cloud prints the coordinates of the point and leaves a marker, and *Shift+Right click* on an existing marker removes it.