

Ruby 3D Depth Camera

User Manual

(v1.0) September 28, 2022



Nerian Vision GmbH
Zettachring 2
70567 Stuttgart
Germany

Email: service@nerian.com
www.nerian.com

Contents

1	Functionality Overview	4
2	Included Parts	4
3	General Specifications	4
3.1	Hardware Details	4
3.2	Stereo Matching	5
3.3	Frame Rates and Resolutions	5
4	Laser Safety	5
5	Mechanical Specifications	6
5.1	Dimensions	6
5.2	Mounting	6
5.3	Temperature	6
6	Physical Interfaces	9
6.1	Interface Overview	9
6.2	Power Supply	10
6.3	GPIO Port	10
6.3.1	Trigger Output	11
6.3.2	Trigger Input	11
6.3.3	Synchronization pulse (PPS)	11
6.4	Reset Button	12
6.5	Status LEDs	12
7	Processing Results	13
7.1	Rectified Images	13
7.2	Disparity Maps	13
7.3	Color Image Projection	16
7.4	Timestamps and Sequence Numbers	16
8	Networking Configuration	17
8.1	IP Configuration	17
8.2	Jumbo Frames	18
9	Configuration	19
9.1	System Status	20
9.2	Presets	21
9.3	Preview	22
9.4	Acquisition Settings	23
9.4.1	Format Settings	23
9.4.2	Frame Rate	24

9.4.3	Exposure Control	24
9.4.4	White Balance Control	24
9.5	Network Settings	25
9.6	Output Channels	26
9.7	Maintenance	27
9.8	Calibration	28
9.8.1	Calibration Board	29
9.8.2	Constraining the image size for calibration	30
9.8.3	Recording Calibration Frames	31
9.8.4	Performing Calibration	31
9.9	Processing Settings	32
9.9.1	Operation Mode	32
9.9.2	Disparity Settings	33
9.9.3	Algorithm Settings	33
9.10	Advanced Auto Exposure and Gain Settings	35
9.10.1	Exposure and Gain	35
9.10.2	Manual Settings	36
9.10.3	ROI Settings	37
9.11	Trigger Settings	37
9.12	Time Synchronization	38
9.13	Reviewing Calibration Results	39
9.14	Auto Re-calibration	41
9.15	Region of Interest	42
9.16	Inertial Measurement Unit	43
9.16.1	Calibration of the inertial measurement unit	44
10	API Usage Information	45
10.1	General Information	45
10.2	ImageTransfer Example	45
10.3	AsyncTransfer Example	47
10.4	3D Reconstruction	48
10.5	Parameters	48
11	Supplied Software	49
11.1	NVCom	49
11.2	GenICam GenTL Producer	50
11.2.1	Installation	50
11.2.2	Virtual Devices	51
11.2.3	Device IDs	52
11.3	ROS Node	52
12	Support	52
13	Warranty Information	53

1 Functionality Overview

Ruby is a stereo-vision-based depth camera. Its two monochrome image sensors record a scene at slightly different viewing positions. By correlating the image data from both image sensors, Ruby can infer the depth of every observed point. The computed depth map is transmitted through 1G Ethernet to a connected computer or another embedded system. An additional color sensor is used for capturing color information, and the color image is automatically aligned to the depth data.

Ruby can perform measurements actively or passively. For active measurements, a laser projector is used to project a pattern onto visible surfaces. This allows objects to be measured even if they have a uniform and textureless appearance.

In situations where the projected pattern cannot be observed, due to bright ambient light, long measurement distances or because the projector is disabled, measurements can still be obtained passively. In case of passive measurements, sufficient surface texture is required for obtaining accurate results.

2 Included Parts

The following parts should be included when ordering a new Ruby 3D depth camera from Nerian Vision Technologies:

- Ruby 3D depth camera
- 12 V DC power supply with interchangeable mains connectors for Europe, North America, United Kingdom and Australia
- Printed user manual
- Ethernet cable, 3 m

If any of these items are missing, then please contact customer support.

3 General Specifications

3.1 Hardware Details

Image sensor	IMX296
Image Resolution	1.5 MP
Sensor format	1/2.9"
Focal length	4.18 mm
Field of View	62.2° × 48.8° (74.0° diagonally)
Aperture	3.0
Pattern projector	Random dot laser (class 1)

Projector wavelength	830 nm
Inertial sensor (IMU)	BNO085
Max. IMU measurement rate	400 Hz (magnetometer: 100 Hz)
Power supply	11.2 – 30 V DC
Power consumption	9 W
Dimensions	130 × 92.5 × 34.5 mm
Weight	ca. 450 g
I/O	Gigabit Ethernet, GPIO
Operating temperature	0 – 40°C
Conformity	CE, FCC, UKCA, RoHS, Laser class 1

3.2 Stereo Matching

Stereo algorithm	Variation of Semi-Global Matching (SGM)
Max resolution	1440 × 1056 pixels
Supported pixel formats	Mono8, Mono12, RGB8
Disparity range	32 to 256 pixels
Frame rate	up to 60 fps
Sub-pixel resolution	4 bits (1/16 pixel)
Post-processing	Consistency check, uniqueness check, gap interpolation, noise reduction, speckle filtering

3.3 Achievable Frame Rates and Image Resolutions

The maximum frame rate that can be achieved depends on the configured image resolution and disparity range. Table 1 provides a list of recommended configurations. This is only a subset of the available configuration space. Differing image resolutions and disparity ranges can be used to meet specific application requirements.

Table 1: Maximum frame rate by image resolution and disparity range.

Disparity Range	Image Resolution		
	720×512	1024×768	1440×1026
128 pixels	60 fps	30 fps	15 fps
256 pixels	n/a	17 fps	8 fps

4 Laser Safety

Ruby contains an infrared laser projector that is not visible to the human eye. The laser complies with the international standards IEC 60825-1:2014 and DIN EN 60825-1:2015 for Class 1. Therefore, the laser is considered *eye-safe* and safety precautions are not necessary.



Figure 1: Laser label on Ruby's bottom side.

The Class 1 laser notice can be found on the product label at the bottom side of the device. This label is depicted in Figure 1.

Any changes or modifications made on the system not expressly approved by the manufacturer could void the user's authority to operate the equipment.

5 Mechanical Specifications

5.1 Dimensions

Figures 2 and 3 show Ruby as seen from different directions. The provided dimensions are measured in millimeters.

5.2 Mounting

The housing of Ruby features two mounting brackets to the sides of the device. Each mounting bracket has two slotted holes, which allows Ruby to be mounted onto a flat surface. The dimensions and placement of the slotted holes are given in Figure 2b.

In addition, Ruby features a 1/4" UNC threaded hole on the bottom side. This allows Ruby to be mounted on a standard camera tripod.

5.3 Temperature

Ruby can be operated without further measures at ambient temperatures between 0°C and 40°C. If operation at a higher ambient temperature is required, additional cooling measures must be taken. Such measures can consist of mounting Ruby onto a thermally conductive surface and/or using a fan to increase the airflow. Please monitor the device temperature sensors (see Section 9.1) when operating Ruby at such elevated ambient temperatures.

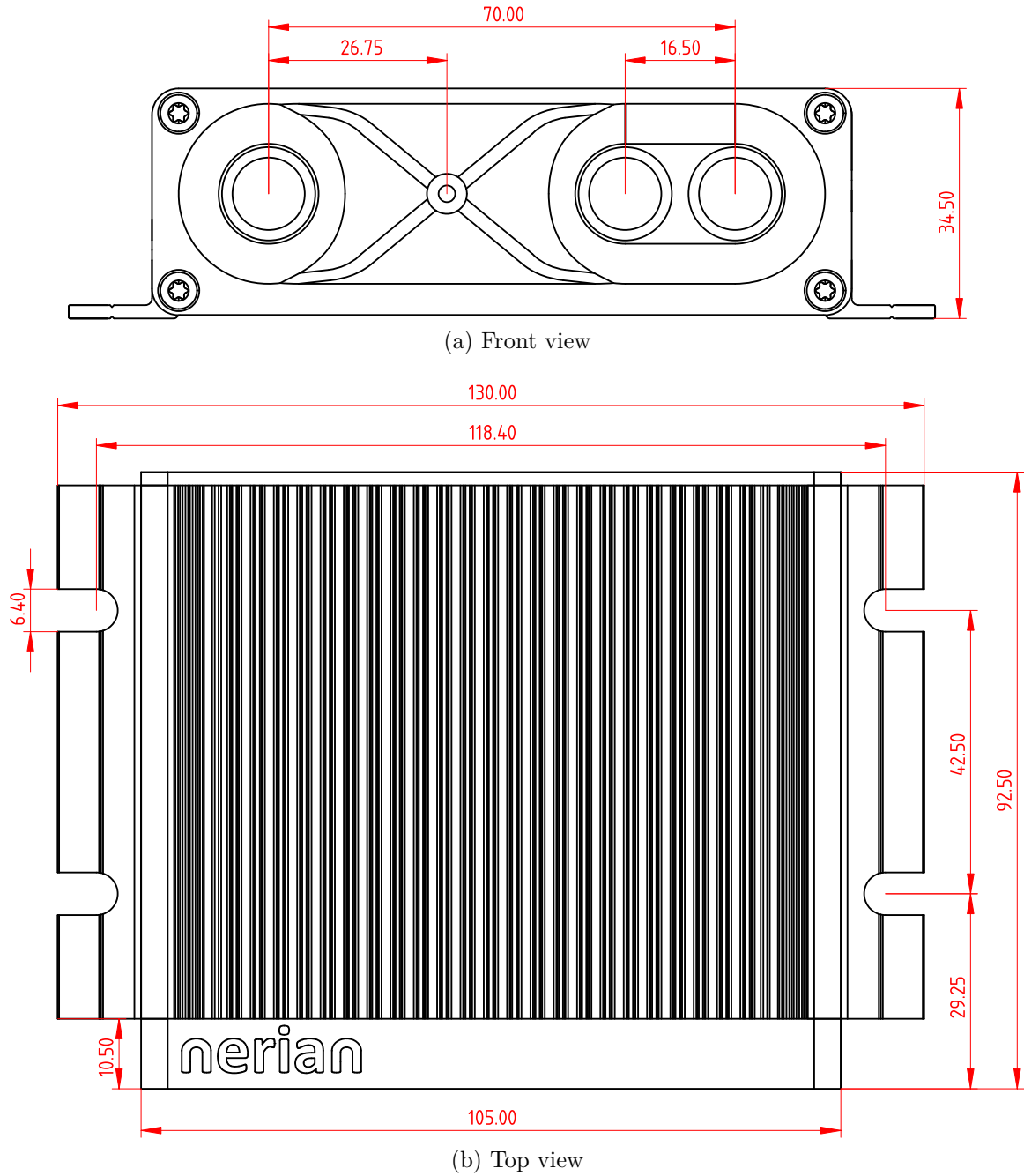


Figure 2: (a) Front and (b) top view of Ruby with dimensions in millimeters.

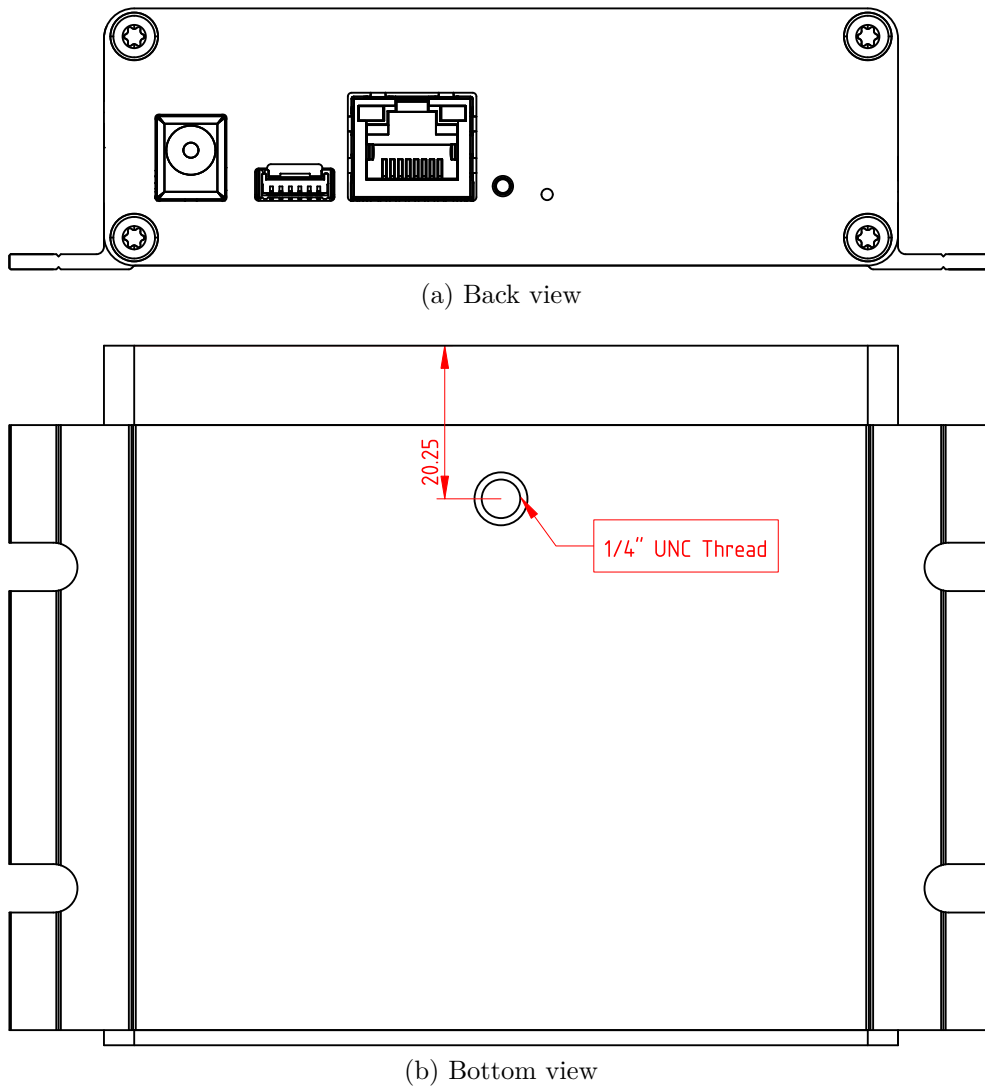


Figure 3: (a) Back and (b) bottom view of Ruby with dimensions in millimeters.

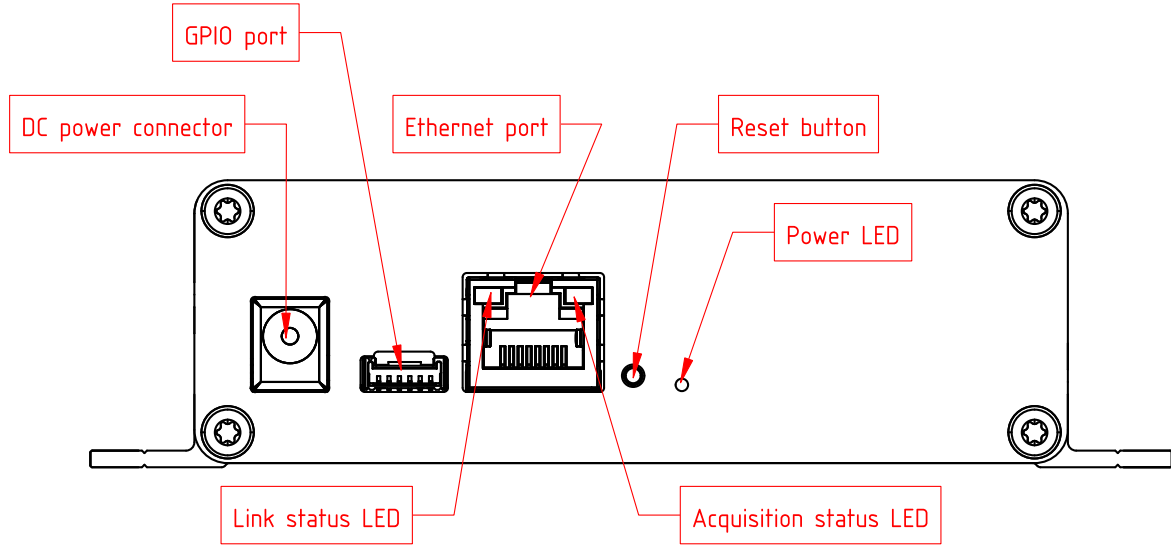


Figure 4: Available interfaces on the backside.

6 Physical Interfaces

6.1 Interface Overview

Figure 4 shows the available physical interfaces on Ruby’s backside. These interfaces are:

DC power connector: Connects to a power supply within the permitted voltage range (see Section 6.2).

GPIO port: Outputs a trigger signal or synchronizes Ruby to an external trigger source. Also functions as an input for the time synchronization pulse (see Section 6.3).

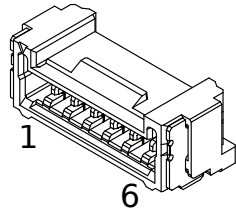
Ethernet port: Port for connecting Ruby to a client computer or another embedded system through a 1G Ethernet. This port is used for delivering processing results and for providing access to the configuration interface.

Reset button: Button for resetting the device firmware back to the factory state (see Section 6.4).

Power LED: A green LED indicating that the device is powered on.

Link status LED (green): Indicates whether an Ethernet link has been established successfully (see Section 6.5)

Acquisition status LED (orange): Indicates the state of image acquisition and reports potential laser failures (see Section 6.5).



Pin	Assignment
1	Trigger input (opto-isolated)
2	Sync input (opto-isolated)
3	Trigger output (opto-isolated)
4	Opto GND
5	+5V
6	GND

Figure 5: Pin assignment of GPIO connector.

6.2 Power Supply

The power connector needs to be connected to the supplied power adapter or another suitable voltage source. When using an alternative power supply, please make sure that the voltage is in the permitted range of 11.2 - 30 V DC. Higher voltages might damage the device. A power supply should be rated for at least 10 W.

The power connector uses a female barrel jack with an internal diameter of 6.5 mm and a pin diameter of 2 mm. The mating connector should have an outer diameter of 5.5 mm. The polarity must be center positive.

6.3 GPIO Port

The GPIO port provides access to the following signals:

- Trigger output
- Trigger input
- Synchronization pulse (PPS)
- +5V DC output

All data input and output signals are connected through opto-couplers. Hence, the Opto GND pin must be used as ground reference for all signals.

In addition to the I/O signals, Ruby provides a 5V DC output, which can deliver a current of up to 100 mA. If the current limit is exceeded, the power output will be switched off.

The GPIO connector uses a female Molex Micro-Lock Plus 505567 series connector. The pin assignment is displayed in Figure 5. The following manufacturer part numbers correspond to matching connectors, and should be used for interfacing:

45111-0606	Matching connector with 600 mm cable
204532-0601	Matching connector without cables

The characteristics of each individual I/O signal is described in the following subsections.

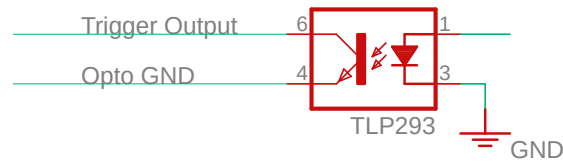


Figure 6: Schematics of trigger output circuit

6.3.1 Trigger Output

In a machine vision application, it might be required to synchronize other sensors or lighting (e.g. a pattern projector) to Ruby's image acquisition. For this purpose, Ruby can output an **open collector** trigger signal on the GPIO pin 3. The signal is isolated through an opto-coupler as shown in the circuit diagram of Figure 6.

The absolute maximum ratings of the opto-coupler are:

Collector-emitter voltage:	max. 80 V
Emitter-collector voltage:	max. 7 V
Collector current:	max. 50 mA
Collector power dissipation:	max. 100 mW

Please see Section 9.11 for further details on how to configure the trigger output.

6.3.2 Trigger Input

Instead of synchronizing other equipment to Ruby's image acquisition, Ruby can also synchronize its image acquisition to an external trigger source, using the trigger input signal on pin 1. The voltage level of an input trigger signal must be between 3.3 V and 24 V. Ruby consumes 2 mA of current on this signal. Please see Section 9.11 for further details on how to configure the trigger input.

6.3.3 Synchronization pulse (PPS)

The synchronization pulse from pins 2 is an input signal that can be used for synchronizing Ruby's internal clock with high precision. Whenever a positive signal edge is received, Ruby can either reset its internal time to 0 or save the current system time and transmit it with the next frame. The voltage of this pulse must be between 3.3 and 24 V. In a typical application, the pulse is generated from a Pulse Per Second (PPS) source.

Please see Section 9.12 for details on how to configure the synchronization pulse and other synchronization methods such as PTP or NTP.

6.4 Reset Button

On the backside of the device there is a recessed reset button. The button is used for resetting Ruby's firmware to the factory state. A reset should be performed if the device becomes unresponsive due to a misconfiguration or firmware corruption. When the reset procedure is started, **all configurations, calibration and installed firmware updates are lost**.

To initiate a reset, gently press the button with a pin, while the device is powered off. Then connect power while keeping the button pressed, and release the button shortly afterwards.

The reset procedure will require several minutes to complete. Once the reset is finished the device will start normally and will be discoverable on the network with the default IP address. You can use the NVCom application (see Section: 11.1) to monitor when the device becomes discoverable after the reset has been completed.

6.5 Status LEDs

The device contains three status LEDs, as depicted in Figure 4:

Power LED (green): The power LED lights up green when the device is powered on.

Link status LED (green): Indicates whether an Ethernet link has been established successfully. If the LED does not light up after connecting an Ethernet cable, then please check the cable for damages and make sure that the remote system (switch or host PC) are operational.

Acquisition status LED (orange): This LED reports the image acquisition state and possible laser failures:

Off: Image acquisition hasn't yet started. This is the case if the device is still booting. Please check the web interface for errors (see Section 9.1), if the LED stays off for more than a few minutes after power-up.

Blinking: Image acquisition has been started successfully and the device is operating as intended.

Constant on: A laser failure has been detected and the laser projector has been switched off. Please contact support for resolving this failure.

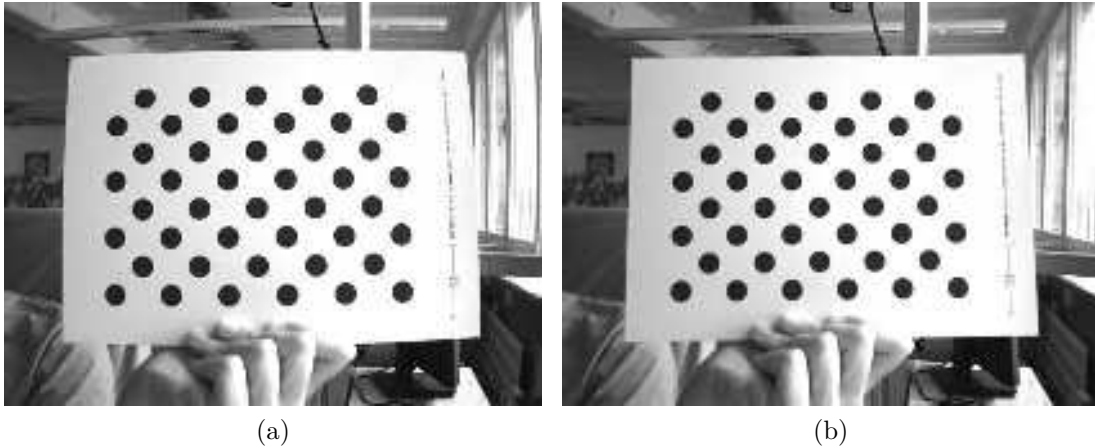


Figure 7: Example for (a) unrectified and (b) rectified camera image.

7 Processing Results

7.1 Rectified Images

Even with Ruby’s precisely aligned image sensors you are unlikely to receive images that match the expected result from an ideal stereo camera. The images are affected by various distortions that result from errors in the optics and sensor placement. Therefore, the first processing step that is performed is an image undistortion operation, which is known as *image rectification*.

Image rectification requires precise knowledge of the camera setup’s projective parameters. These can be determined through camera calibration. Please refer to Section 9.8 for a detailed explanation of the camera calibration procedure. Ruby will be shipped pre-calibrated and a re-calibration will typically not be necessary over the device’s lifespan.

Figure 7a shows an example camera image, where the camera was pointed towards a calibration board. The edges of the board appear slightly curved, due to radial distortions caused by the camera’s optics. Figure 7b shows the same image after image rectification. This time, all edges of the calibration board appear perfectly straight.

7.2 Disparity Maps

The stereo matching results are delivered in the form of a *disparity map* from the perspective of the left monochrome camera. The disparity map associates each pixel in the left camera image with a corresponding pixel in the right camera image. Because both images were previously rectified to match an ideal stereo camera geometry, corresponding pixels should only differ in their horizontal coordinates. The disparity map thus only encodes a *horizontal coordinate difference*.

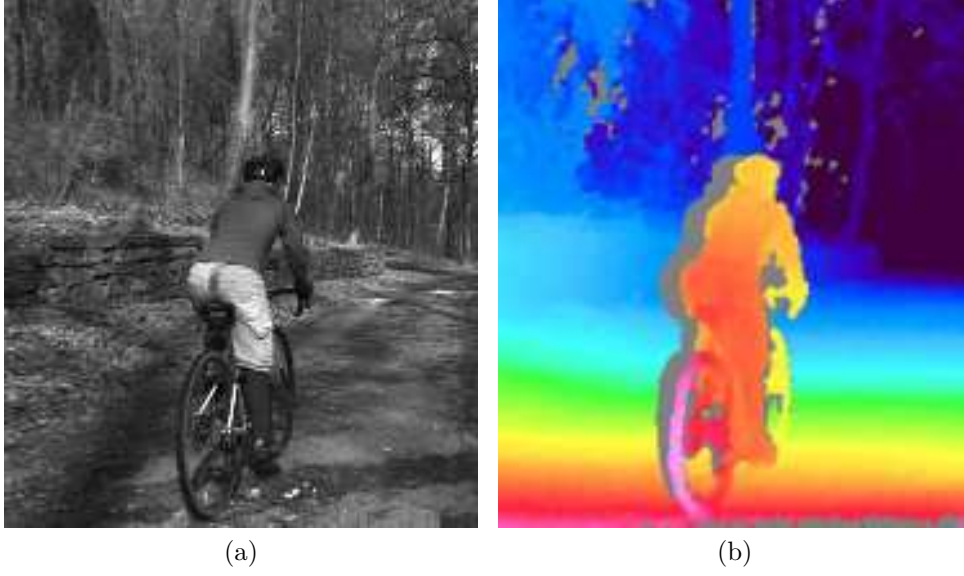


Figure 8: Example for (a) left camera image and corresponding disparity map.

Examples for a left camera image and the corresponding disparity map are shown in Figures 8a and 8b. Here the disparity map has been color coded, with blue hues reflecting small disparities, and red hues reflecting large disparities. As can be seen, the disparity is proportional to the inverse depth of the corresponding scene point.

The *disparity range* specifies the image region that is searched for finding pixel correspondences. A large disparity range allows for very accurate measurements, but causes a high computational load, and thus lowers the achievable frame rate. Ruby supports a configurable disparity range (see Section 9.9), which allows the user to choose between high-precision or high-speed measurements.

It is possible to transform the disparity map into a set of 3D points. This can be done at a correct metric scale if the system has been calibrated properly. The transformation of a disparity map into a set of 3D points requires knowledge of the disparity-to-depth mapping matrix Q , which is computed during camera calibration and transmitted by Ruby along with each disparity map. The 3D location $\begin{pmatrix} x & y & z \end{pmatrix}^T$ of a point with image coordinates (u, v) and disparity d can be reconstructed as follows:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \frac{1}{w} \cdot \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}, \text{ with } \begin{pmatrix} x' \\ y' \\ z' \\ w \end{pmatrix} = Q \cdot \begin{pmatrix} u \\ v \\ d \\ 1 \end{pmatrix}$$

When using the Q matrix provided by Ruby, the received coordinates will be measured in meters with respect to the coordinate system depicted in Fig-

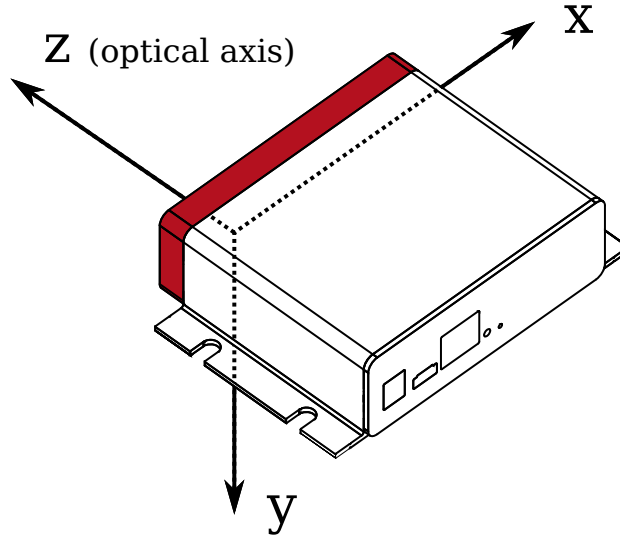


Figure 9: Coordinate system used for 3D reconstruction.

ure 9. Here, the origin matches the lens' center of projection (the location of the aperture in the pinhole camera model) for the left monochrome camera. An efficient implementation of this transformation is provided with the available API (see Section 10.4).

Ruby computes disparity maps with a disparity resolution that is below one pixel. Disparity maps have a bit-depth of 12 bits, with the lower 4 bits of each value representing the fractional disparity component. **It is thus necessary to divide each value in the disparity map by 16, in order to receive the correct disparity magnitude.**

Ruby applies several post-processing techniques in order to improve the quality of the disparity maps. Some of these methods detect erroneous disparities and mark them as invalid. Invalid disparities are set to `0xFF`, which is the highest value that can be stored in a 12-bit disparity map. In the example disparity map from Figure 8b, invalid disparities are depicted as grey.

Please note that there is usually a stripe of invalid disparities on the left image border of a disparity map. This behavior is expected as the disparity map is computed from the perspective of the left camera. Image regions on the left edge of the left camera image cannot be observed by the right camera, and therefore no valid disparity can be computed. The farther left an object is located, the farther away it has to be, in order to also be visible to the right camera. Hence, the full depth range can only be observed for left image pixels with a horizontal image coordinate $u \geq d_{max}$.

Likewise, invalid disparities can be expected to occur to the left on any foreground object. This shadow-like invalid region is caused by the visible background being occluded in the right camera image but not in the left camera image. This effect is known as the *occlusion shadow* and is clearly visible in the provided example image.

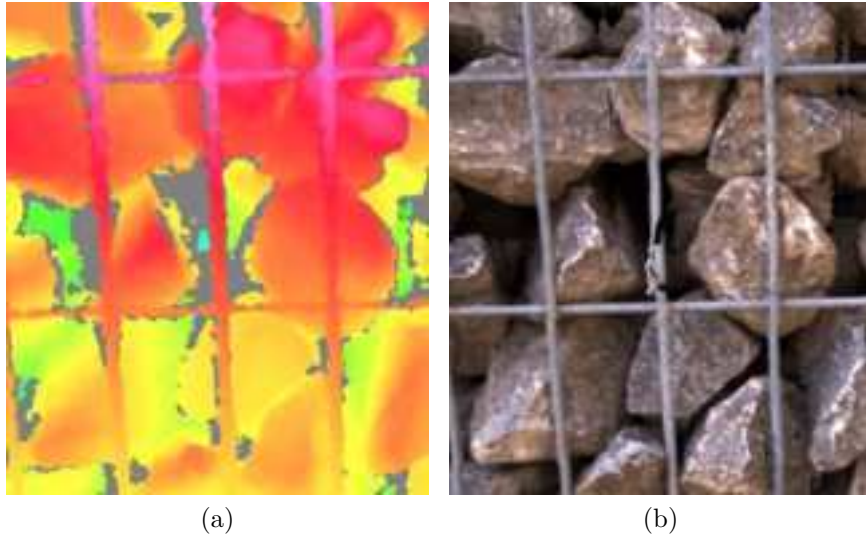


Figure 10: Example for (a) disparity map and (b) projected color image with artifact.

7.3 Color Image Projection

The left monochrome sensor is used as reference camera for depth computation. Even though the color sensor is placed right next to it, there will be a parallax (an apparent optical shift) between the color image and the disparity map / left monochrome image.

This shift can be compensated by projecting the color image back into the view of the reference camera. Once this projection is performed, corresponding image points between the left monochrome image, disparity map and color image will all have identical image coordinates, and all three images can be directly overlaid.

Ruby is capable of performing this projection automatically. The projection depends on the depth measurements and is unfortunately not perfect. This means that some visual artifacts are to be expected. The amount of artifacts depends strongly on the depth measurement quality. Particularly object edges can be affected by artifacts.

A magnified subsection of an example color image and depth map that show this effect can be seen in Figure 10. In cases where a parallax between the depth measurements and color image are acceptable, artifacts can be avoided by disabling this projection. For more details, please refer to Section 9.6.

7.4 Timestamps and Sequence Numbers

Each set of images that is transmitted by Ruby, also includes a timestamp and a sequence number. The timestamp is measured with microsecond accuracy and is set to the time at which the image sensors started exposing a frame.

Hence the exposure time should always be considered when attempting to measure the sensor delay.

As explained in Sections 6.3.3 and 9.12, it is possible to synchronize Ruby's internal clock to an external signal or a time server. This directly affects the produced time stamps. When synchronized to a time server, time stamps are measured in microseconds since 1 January 1970, 00:00:00 UTC. If no synchronization is performed, the internal clock is set to 0 at powered up. When synchronizing to an external PPS signal, as explained in Section 6.3.3, the clock is set to 0 on the incoming rising signal edge.

Please note that synchronizing to a PPS signal also produces negative timestamps. This happens when a synchronization signal is received while Ruby is processing an already captured image pair. The negative timestamp is then the time difference between the reception of the synchronization signal and the time of capturing the current image pair.

8 Networking Configuration

It is recommended to connect Ruby directly to the host computer's ethernet port, without any switches or hubs in between. This is because Ruby produces very high-throughput network data, which might lead to packet loss when using network switches that cannot meet the required performance. It must be ensured that the host computer's network interface can handle an incoming data rate of 900 MBit/s.

The necessary network configuration settings for the host computer are described in the following subsections.

8.1 IP Configuration

By default, Ruby will use the IP address 192.168.10.10 with subnet mask 255.255.255.0. If a DHCP server is present on the network, however, it might assign a different address to Ruby. In this case please use the provided NVCom software for discovering the device (see Section 11.1).

If no other DHCP server is present on the network, Ruby will start its own DHCP server. This means that if your computer is configured to use a dynamic IP address the computer will automatically receive an IP address in the correct subnet and no further configuration is required.

If your computer is not configured to use a dynamic IP address or Ruby's integrated DHCP server is disabled, then you need to configure your IP address manually. For Windows 10 please follow these steps:

1. Click Start Menu > Settings > Network & Internet > Ethernet > Change adapter options.
2. Right-click on the desired Ethernet connection.

3. Click ‘Properties’
4. Select ‘Internet Protocol Version 4 (TCP/IPv4)’.
5. Click ‘Properties’.
6. Select ‘Use the following IP address’.
7. Enter the desired IP address (192.168.10.xxx).
8. Enter the subnet mask (255.255.255.0).
9. Press OK.

For Linux, please use the following commands to temporarily set the IP address 192.168.10.xxx on network interface eth0:

```
sudo ifconfig eth0 192.168.10.xxx netmask 255.255.255.0
```

8.2 Jumbo Frames

For maximum performance, Ruby should be configured to use Jumbo Frames (see Section 9.5). By default, Jumbo Frame support might not be enabled in the shipped configuration, as this requires an appropriate configuration of the host computer’s network interface.

If Ruby is accessible via the web interface and discovered in the devices list (e.g. in NVCom, see Section 11.1), but no image data is received (0 fps), this might indicate that Jumbo Frames are activated in Ruby, but the network connection of the respective client computer is not properly configured to accept them.

In order to activate Jumbo Frame support in Windows 10, please follow these steps:

1. Open ‘Network and Sharing Center’
2. Open the properties dialog of the desired network connection
3. Press the button ‘Configure...’
4. Open the ‘Advanced’ tab
5. Select ‘Jumbo Packet’ and choose the desired packet size (see Figure 11)

Please note that unlike for Linux, some Windows network drivers also count the 14-byte Ethernet header as part of the packet size. **When configuring Ruby to use a 9000 bytes MTU, a Windows computer might require a 9014 bytes packet size.**

On Linux, Jumbo Frame support can be activated by setting a sufficiently large MTU, through the `ifconfig` command. For configuring a 9000 bytes MTU for interface eth0, please use the following command line:

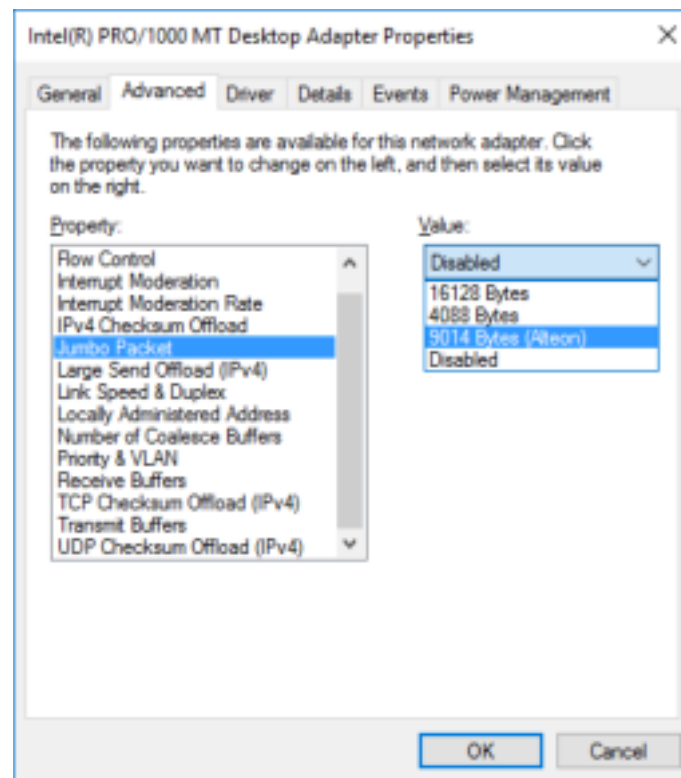


Figure 11: Jumbo Frames configuration in Windows

```
> sudo ifconfig eth0 mtu 9000
```

Please be aware that the interface name might be different from `eth0`, especially in newer Linux releases.

The MTU is assigned automatically according to the Ruby Jumbo Frame settings whenever a Linux computer receives configuration from an active Ruby DHCP server (see Section 9.5). On Windows, automatic MTU assignment does not work, as Windows does not support this feature.

9 Configuration

Ruby is configured through a web interface, which can be reached by entering its IP address into your browser. The default address is `http://192.168.10.10` but if a DHCP server is present on the network, it might assign a different address to Ruby (see Section 8.1). In this case please use the provided NVCom software for discovering the device (see Section 11.1).

If Ruby has just been plugged in, it will take several seconds before the web interface is accessible. **For using the web interface, you require a browser with support for HTML 5.** Please use a recent version of one of the major browsers, such as Chrome, Firefox, Safari, or Edge.

The web-interface is divided into two sections: *General Settings* and *Ad-*

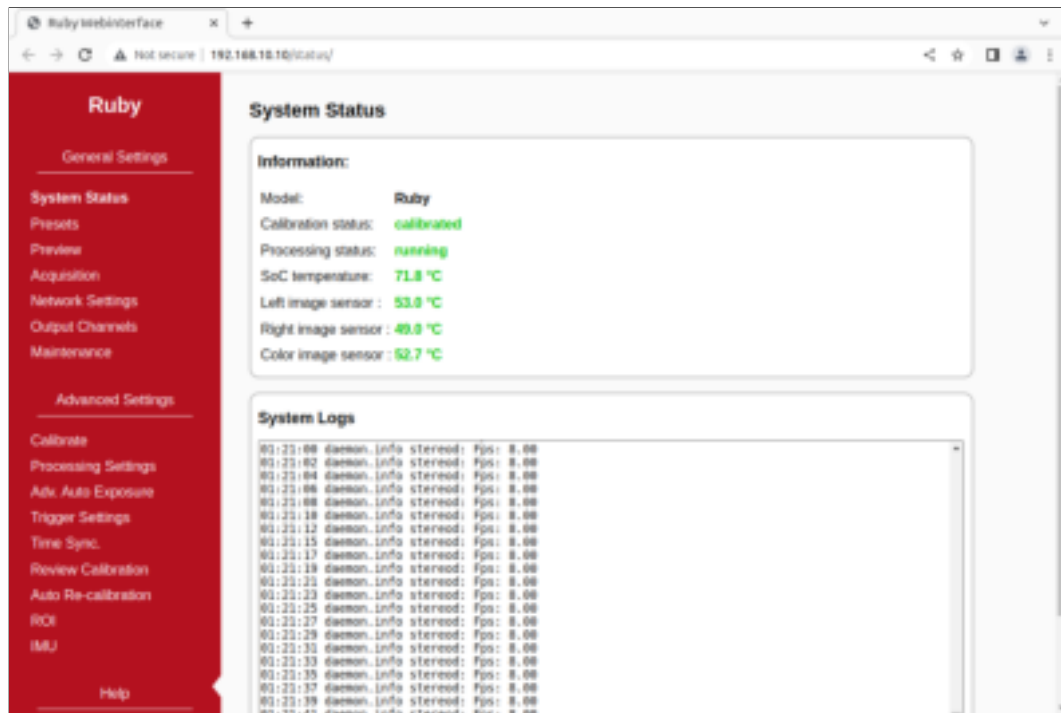


Figure 12: Screenshot of configuration status page.

vanced Settings. The general settings pages contain the most commonly adjusted parameters. Modifying only these parameters should be sufficient for most applications. Less commonly adjusted parameters that might be relevant for very specific applications can be found on the advanced settings pages.

9.1 System Status

The first page that you see when opening the web interface is the ‘*system status*’ page that is shown in Figure 12. On this page, you can find the following information:

Model: The model name for your device.

Calibration status: Provides information on whether the system has been correctly calibrated.

Processing status: Indicates whether the image processing sub-system has been started. If this is not the case, then there might be a configuration problem, or a system error might have occurred. Please consult the system logs in this case. The image processing sub-system will be started immediately once the cause of error has been resolved.

SOC temperature: The temperature of the central System-on-Chip (SoC) that performs all processing tasks. The maximum operating temperature



Figure 13: Screenshot of configuration presets page.

for the employed SoC is at 100 °C. A green-orange-red color-coding is applied to signal good, alarming and critical temperatures.

Left/right/color image sensor: Chip temperatures for the left, right and color image sensors. The maximum operating temperature for the image sensors is 75 °C. Like for the SOC temperature, a green-orange-red color coding is applied.

System logs: List of system log messages sorted by time. In regular operation, you will find information on the current system performance. In case of errors, the system logs contain corresponding error messages.

9.2 Presets

Different configuration presets are available for selected combinations of image resolution and frame rate. The use of a preset is highly recommended, as it will guarantee an optimal use of Ruby's performance.

Figure 13 shows the *presets* web-interface page. Loading a preset will only modify the parameters that are relevant for a given configuration. Other parameters will not be modified. If all parameters should be set to the preferred default value, it is recommended to first perform a configuration reset (see Section 9.7) and then load the desired preset afterwards.

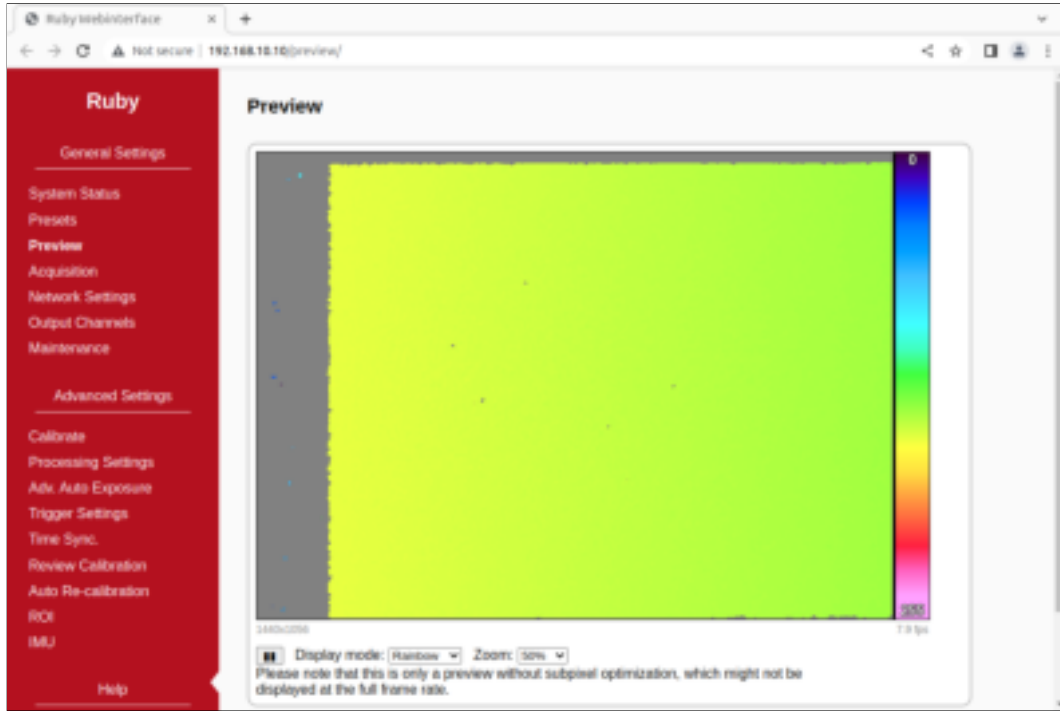


Figure 14: Screenshot of configuration preview page.

9.3 Preview

The *preview* page, which is shown in Figure 14, provides a live preview of the currently computed disparity map. Please make sure that your network connection supports the high bandwidth that is required for streaming video data (see Section 8.2). For using the preview page, you require a direct network connection to Ruby. An in-between proxy server or a router that performs network address translation (NAT) cannot be used.

When opening the preview page, Ruby stops transferring image data to any other host. The transfer is continued as soon as the browser window is closed, the user presses the pause button below the preview area, or if the user navigates to a different page. Only one open instance of the preview page, or any other page that is streaming video data to the browser, is allowed at a time. If attempted to open more than once, only one instance will receive data.

The preview that is displayed in the browser does not reflect the full quality of the computed disparity map. In particular, the frame rate is limited to 20 fps and sub-pixel accuracy is not available. To receive a full-quality preview, please use the NVCom application, which is described in Section 11.1.

Different color-coding schemes can be selected through the drop-down list below the preview area. A color scale is shown to the right, which provides information on the mapping between colors and disparity values. The possible

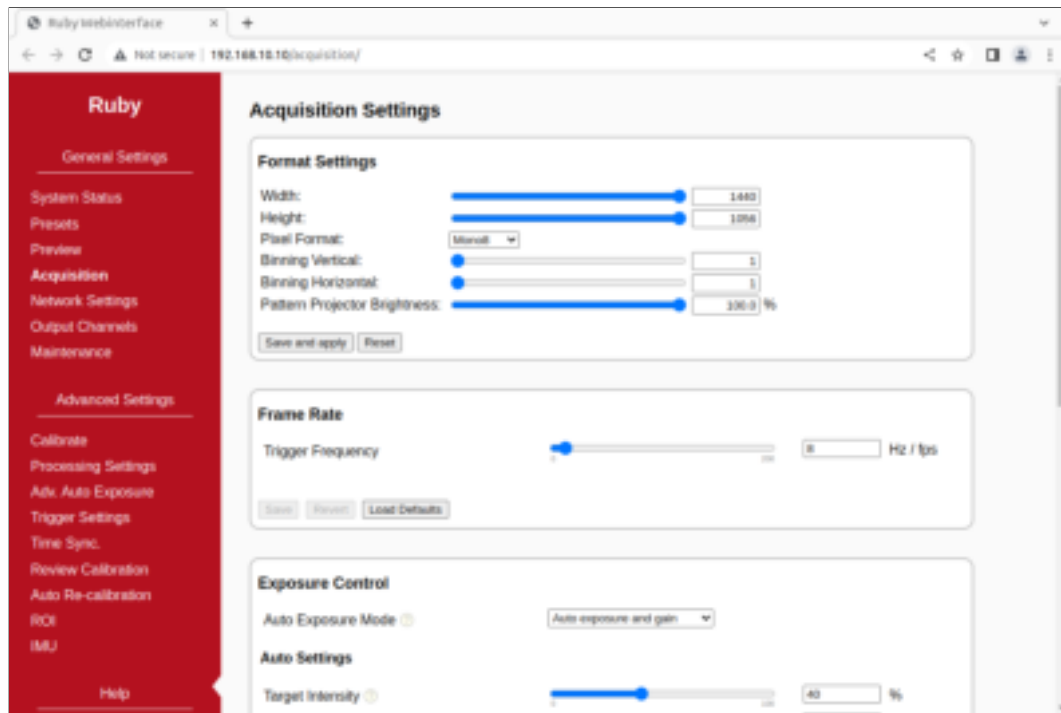


Figure 15: Screenshot of configuration page for acquisition settings.

color schemes are:

Rainbow: A rainbow color scheme with low wavelengths corresponding to high disparities and high wavelengths corresponding to low disparities. Invalid disparities are depicted in gray.

Red / blue: A gradient from red to blue, with red hues corresponding to high disparities and blue hues corresponding to low disparities. Invalid disparities are depicted in black.

Raw data: The raw disparity data without color-coding. The pixel intensity matches the integer component of the measured disparity. Invalid disparities are displayed in white.

9.4 Acquisition Settings

The most relevant parameters for image acquisition are listed on the *acquisition settings* page that is shown in Figure 15. This page is divided into three distinct areas.

9.4.1 Format Settings

This section covers all settings related to the image format. Rather than modifying the format settings individually, we recommended to use a preset

(see Section 9.2), and only change individual settings if needed. This will ensure that Ruby's imaging and processing capabilities are used optimally.

Please note that the *apply* button must be pressed in order for any configuration changes to become effective. The available settings are:

Width:	Width in pixels of the selected Region-Of-Interest (ROI). Also see Section 9.15 for more ROI options.
Height:	Height in pixels of the selected ROI.
Pixel Format:	Desired pixel encoding mode. Available settings are 8-bit mono (Mono8) or 12-bit mono (Mono12P).
Binning Horizontal:	Number of horizontal photosensitive cells that are combined for one image pixel.
Binning Vertical:	Number of vertical photosensitive cells that are combined for one image pixel.
Pattern Projector Brightness:	Brightness of the pattern projector specified in percent. 100% indicates full brightness, whereas 0% turns the projector off completely.

9.4.2 Frame Rate

The frame rate at which Ruby records images can be freely configured. The maximum frame rate that can be achieved depends on the chosen image resolution, disparity range, pixel format and network interface. If you set a frame rate that is higher than the achievable maximum, then this can result in an irregular image acquisition or no frames being acquired. It is recommended to first select a preset (see Section 9.2) with the desired resolution, and then only lower the frame rate if needed.

9.4.3 Exposure Control

Ruby will automatically control the sensor exposure and gain to match a given average intensity, which can be selected in the '*exposure control*' area. If an automatic adjustment is not desired, then the user can alternatively specify a manual exposure time and gain setting. More advanced exposure and gain options are available on the '*advanced auto exposure and gain settings*' page (see Section 9.10).

9.4.4 White Balance Control

Ruby supports automatic or manual white balancing; the red and blue color balance factors can be controlled. This functionality can be configured in the '*white balance control*' area. In the default white balance mode, '*automatic (gray world)*', the color channel balance settings are adjusted in real time, based on a heuristic estimation of illumination color from the image data. In the '*manual*' white balance mode, the algorithm is disabled, and the red and

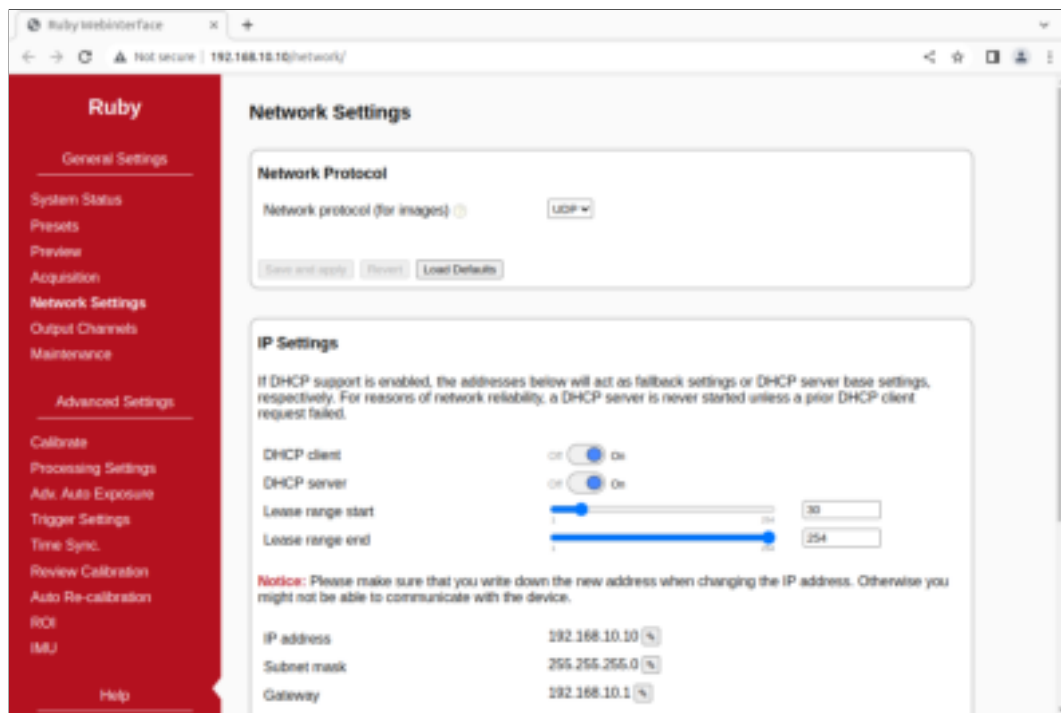


Figure 16: Screenshot of configuration page for network settings.

blue balance factors can be adjusted manually. The currently effective balance factors are also shown in the area.

9.5 Network Settings

The ‘*network settings*’ page, which is displayed in Figure 16, is used for configuring all network related parameters. Ruby can query the network configuration automatically via DHCP client requests, which are enabled by default to aid switching between existing network setups. Ruby devices in a network that assigns IP settings through DHCP are easily discovered and accessed via the device discovery API and also the NVCom utility (Section 11.1). If no DHCP servers are present, Ruby uses its static IP settings as a fallback.

DHCP client support can be disabled if fixed IP settings are desired and the device will not be switched between different networks. In this case, the IP settings in this section are used as static values.

Ruby also contains a fallback DHCP server. It is enabled by default but only launched when a prior DHCP client request failed. This means that no DHCP server is ever launched if DHCP client support is turned off, to ensure that Ruby will never compete with an existing DHCP server. The Ruby DHCP server uses the IP address settings as a base; the lease range is always in the /24 subnet of the IP address.

In the ‘*IP settings*’ section, you can disable or enable the DHCP compo-

nents and specify an IP address, subnet mask and gateway address, which are used as static configuration or fallback configuration depending on the DHCP settings. When changing the IP settings, please make sure that your computer is in the same subnet, or that there exists a gateway router through which data can be transferred between both subnets. Otherwise you will not be able to access the web interface anymore and you might be forced to perform a firmware reset (see Section 6.4).

In the ‘*network protocol*’ section, you can choose the underlying network protocol that shall be used for delivering the computation results to the client computer. The possible options are *TCP* and *UDP*. Due to the high-bandwidth real time data we recommend using *UDP*.

In order to obtain the best possible performance, jumbo frames support should be activated in the ‘*jumbo frames*’ section. Before doing so, however, you must make sure that jumbo frames support is also enabled for your client computer’s network interface. Details on how to enable jumbo frame support on your computer can be found in Section 8.2 on page 18. For Linux client computers, the jumbo frames (MTU) setting is automatically applied when receiving configuration from an active Ruby DHCP server. Please note that in this case changing the Ruby Jumbo Frames mode or MTU Size necessitates new DHCP leases to propagate the setting (e.g. by unplugging and re-inserting the network cable).

9.6 Output Channels

The active output channels can be configured on the ‘*output channels*’ page. An output channel is an image data stream that is transmitted over the network. The following output channels are available:

- Left camera output
- Disparity output
- Right camera output
- Color camera output

If the operation mode (see Section 9.9) is set to *stereo matching* (the default) or *rectify*, then the image data of all output channels are rectified (see Section 7.1 for details). If the operation mode is set to *pass through*, however, the camera images will be transmitted without modifications.

As described in section 7.3, the image of the color camera can be projected to the view of the left camera. This projection can be activated by selecting the corresponding option for the ‘*color output mode*’ parameter.

Please note that increasing the number of active output channels also increases the network load and might result in a reduced frame rate. All performance specifications given in this document refer to a configuration with only the color and disparity output channel activated.

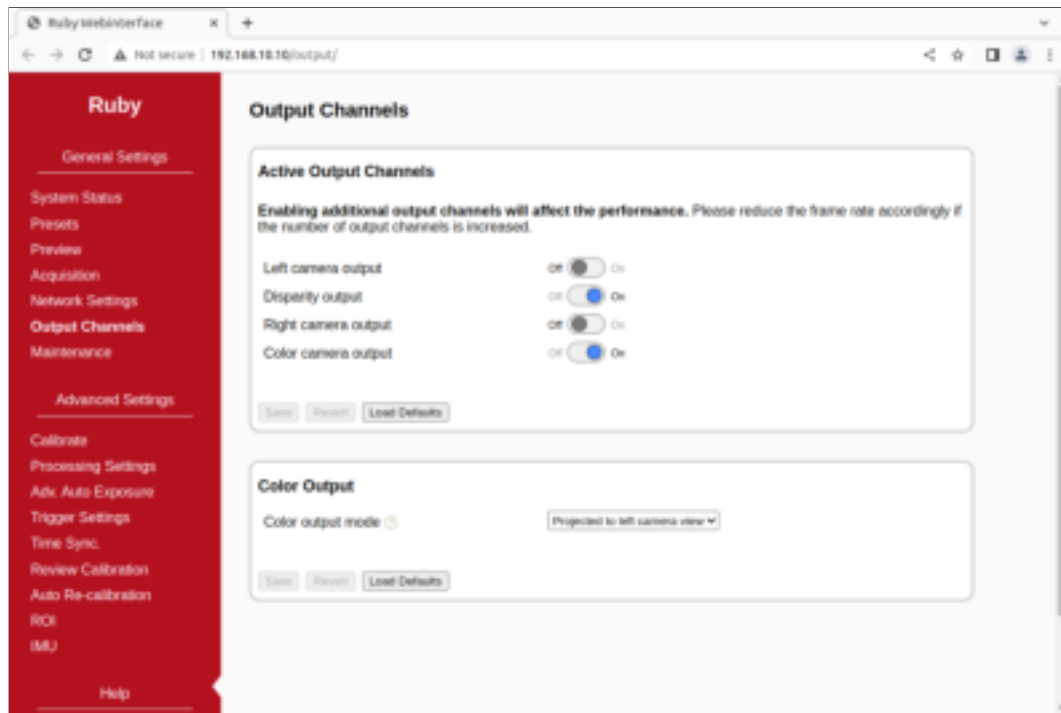


Figure 17: Screenshot of output channels configuration page.

9.7 Maintenance

On the *maintenance* page that is shown in Figure 18, you can download a file that contains the current device configuration and the system logs, by pressing the *download* link. In case of technical problems please include this file in your support request, such that your device configuration can be reproduced and that system problems can be investigated.

A downloaded configuration file can be re-uploaded at a later point in time. This allows for a quick switching between different device configurations. In order to upload a configuration, please select the configuration file and press the *upload* button. Please be aware that uploading a different configuration might modify the IP address of the device. In order to avoid a faulty configuration state, please only upload configurations that have previously been downloaded through the web interface.

If you are experiencing troubles with your current device configuration, you can reset all configuration settings to the factory defaults, by pressing the *reset* button. Please note that this will also reset the network configuration, which might lead to a change of Ruby's IP address.

If Ruby shows signs of erroneous behavior, it is possible to reboot the device by pressing the '*reboot now*' button. It will take several seconds until a reboot is completed and Ruby is providing measurement data again. Please use this function as an alternative to a power cycle, if the device cannot be

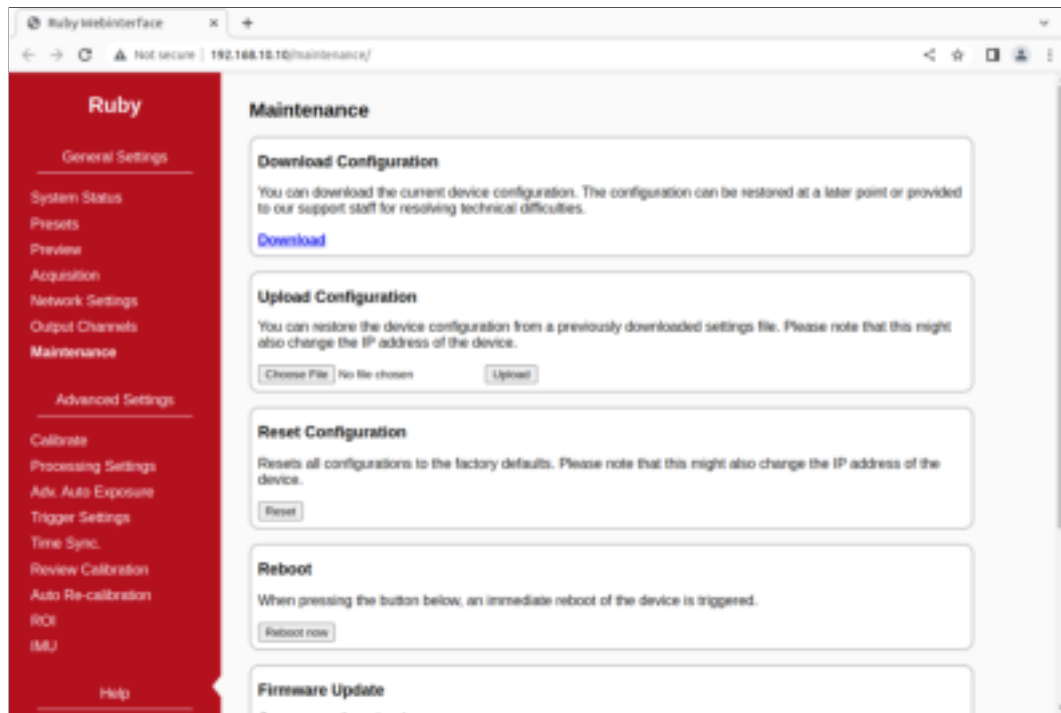


Figure 18: Screenshot of configuration maintenance page.

easily accessed.

The maintenance page further allows you to perform firmware updates. Use this functionality only for firmware files that have officially been released by Nerian Vision Technologies. To perform a firmware update, select the desired firmware file and press the *update* button. The update process will take several seconds. **Do not unplug the device, reload the maintenance page or re-click the update button while performing firmware updates.** Otherwise, this might lead to a corrupted firmware state. Once the update has been completed the device will automatically perform a reboot with the new firmware version. The device configuration is preserved during firmware updates, but some updates might require you to adjust specific settings afterwards.

9.8 Calibration

Ruby is shipped pre-calibrated and a user calibration is typically not required throughout the device's lifetime. However, should you experience a reduction in measurement quality and density, you can correct for potential optical misalignments by performing a re-calibration. In this case the calibration page, which is shown in Figure 19, shall be used.

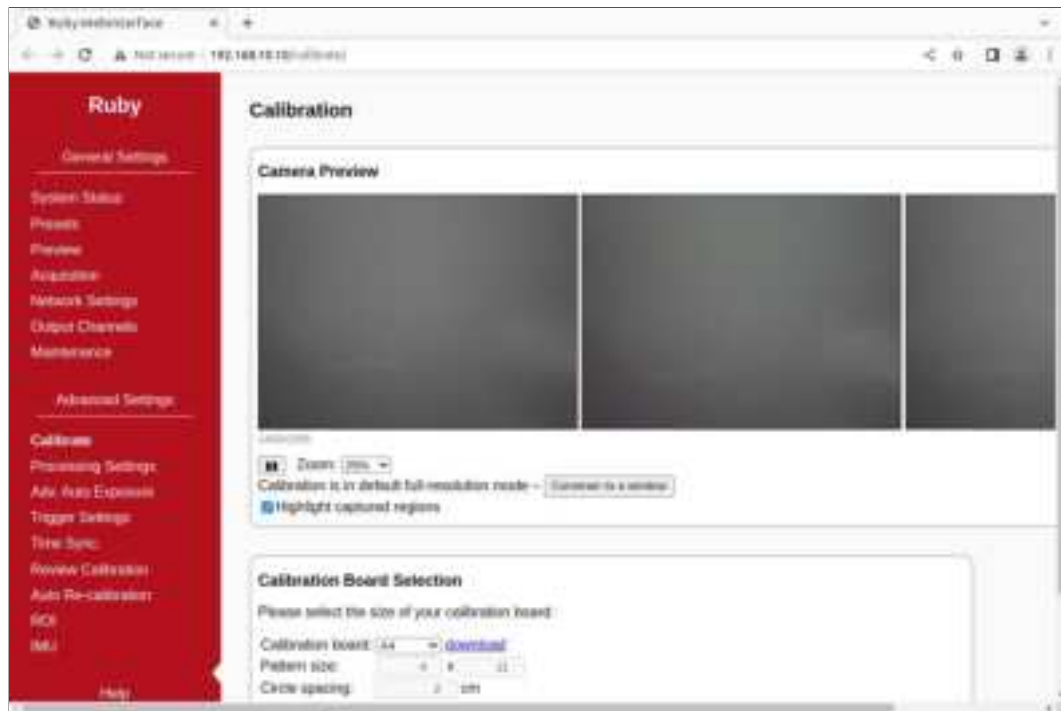


Figure 19: Screenshot of configuration page for camera calibration.

9.8.1 Calibration Board

You require a calibration board, which is a flat panel with a visible calibration pattern on one side. The pattern that is used by Ruby consists of an asymmetric grid of black circles on a white background, as shown in Figure 20.

When opening the calibration page, you will first need to specify the size of the calibration board, which you are going to use in the calibration process. Please make sure to select the correct size, as otherwise the calibration results cannot be used for 3D reconstruction with a correct metric scale (see Section 7.2).

The pattern can be downloaded directly from this page. Simply select the desired pattern size in the ‘*calibration board*’ drop-down list, and click the download link.

Should you require a calibration board with a custom size, then you can select *custom* from the ‘*calibration board*’ drop-down list. This allows you to enter the calibration board details manually. The first dimension of the *pattern size* is the number of circles in one grid column. This number must be equal for all columns of the circles grid.

The number of circles per row is allowed to vary by 1 between odd and even rows. The second dimension is thus the sum of circles in two consecutive rows. All downloadable default calibration patterns have a size of 4×11 .

The last parameter that you have to enter when using a custom calibration

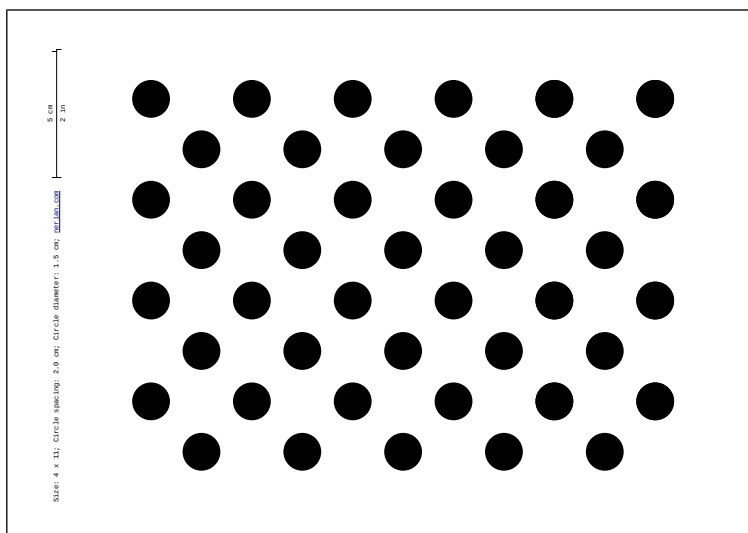


Figure 20: Calibration board used by Ruby.

board is the *circle spacing*. This is the distance between the centers of two neighboring circles. The distance must be equal in horizontal and vertical direction for all circles.

Once the correct board size has been specified, please click on the continue button to proceed with the calibration process.

9.8.2 Constraining the image size for calibration

By default, the calibration process will run on the full sensor area, with the maximum valid image size available for the currently active image format and acquisition settings. This is recommended for most setups, since a smaller Region of Interest can be selected at any time post-calibration (see Section 9.15). For special setups, for example if the image circle of a lens is smaller than the image sensor area, it is necessary to constrain the relevant sensor region prior to the initial calibration.

By pressing the ‘*constrain to a window*’ button in the bottom of the ‘*camera preview*’ area, a centered overlay frame is displayed, which can be resized by dragging. If applied, calibration will switch to constrained-region mode. Calibration can be returned to the default operation by pressing the ‘*reset to full-resolution*’ button.

When the calibration process has been successfully completed with a constrained region, this will reduce the default output size (and maximum available Region-of-Interest size) from the maximum valid image size to the selected one, effectively excluding any areas that are outside the calibrated sensor region.

9.8.3 Recording Calibration Frames

A live preview of all image sensors is displayed in the ‘*camera preview*’ area. Unless the calibration region has been constrained as outlined above, the camera resolution during calibration is set to the maximum valid image size for the currently active image format and acquisition settings. Make sure that the calibration board is fully visible in all camera images and then press the ‘*capture single frame*’ button in the *control* section. Repeat this process several times while moving either the camera or the calibration board.

The calibration board must be recorded at multiple different positions and orientations. A green overlay will be displayed in the preview window for all locations, where the board has previously been detected. You should vary the distance of the board and make sure that you cover most of the field of view of all cameras.

The more frames you record, the more accurate the computed calibration will be. However, more frames also cause the computation of the calibration parameters to take longer. Ruby supports the recording of up to 40 calibration frames. We recommend using at least 20 calibration frames in order to receive accurate results.

The recording of calibration frames can be simplified by activating the ‘*auto capture*’ mode. In this mode, a new calibration frame is recorded in fixed *capture intervals*. You can enter the desired interval in the auto capture section and then press the ‘*start auto capture*’ button. If desired, an audible sound can be played to signal the countdown and the recording of a new frame. Auto capture mode can be stopped by pressing the ‘*stop auto capture*’ button.

A small preview of each captured calibration frame is added to the ‘*captured frames*’ section. The frames are overlaid with the detected positions of the calibration board circles. You can click any of the preview images to see the calibration frame at its full resolution. An example for a calibration frame with a correctly detected calibration board is shown in Figure 21. If the calibration board was not detected correctly or if you are unhappy with the quality of a calibration frame, then you can delete it by clicking on the ×-symbol.

9.8.4 Performing Calibration

Once you have recorded a sufficient number of calibration frames, you can initiate the calibration process by pressing the *calibrate* button in the *control* section. The time required for camera calibration depends on the number of calibration frames that you have recorded. Calibration will usually take several minutes to complete. If calibration is successful then you are immediately redirected to the ‘*review calibration*’ page.

Calibration will fail if the computed vertical or horizontal pixel displacement exceeds the allowed range for any image point. The most common causes for calibration failures are:

- Insufficient number of calibration frames.

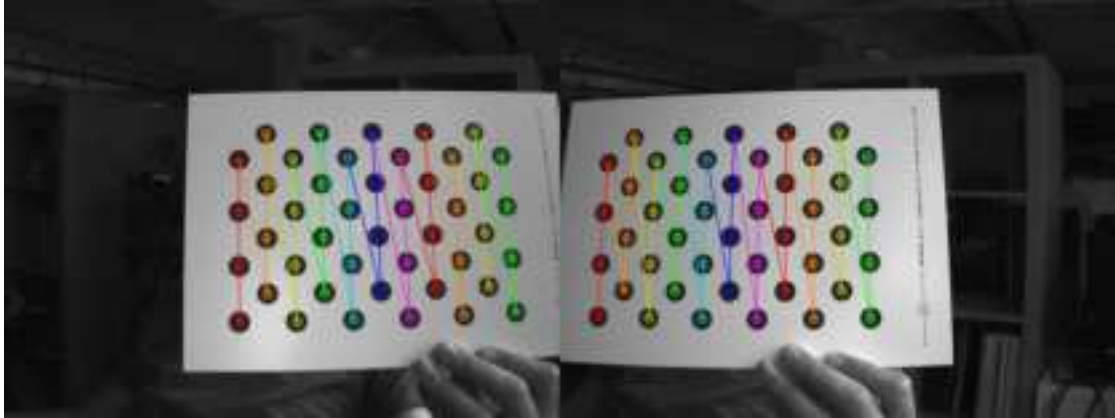


Figure 21: Example calibration frame with detected calibration board.

- Poor coverage of the field of view with the calibration board.
- Lenses with strong geometric distortions.
- Lenses with unequal focal lengths.
- Frames with calibration board misdetections.

Should calibration fail, then please resolve the cause of error and repeat the calibration process. If the cause of error is one or more erroneous calibration frames, then you can delete those frames and re-press the *calibrate* button. Likewise, in case of too few calibration frames, you can record additional frames and restart the calibration computation.

9.9 Processing Settings

9.9.1 Operation Mode

The major processing parameters can be changed on the ‘*processing settings*’ page, which is shown in Figure 22. The most relevant option is the operation mode, which can be set to one of the following values:

Pass through: In this mode Ruby forwards the imagery of all image sensors without modification. This mode is intended for reviewing the image data before any processing is applied.

Rectify: In this mode Ruby transmits the rectified images of all image sensors. This mode is intended for verifying the correctness of the image rectification.

Stereo matching: This is the default mode, in which Ruby performs the actual stereo image processing (stereo matching). Ruby transmits the disparity map and, depending on the output channels configuration, the rectified images.

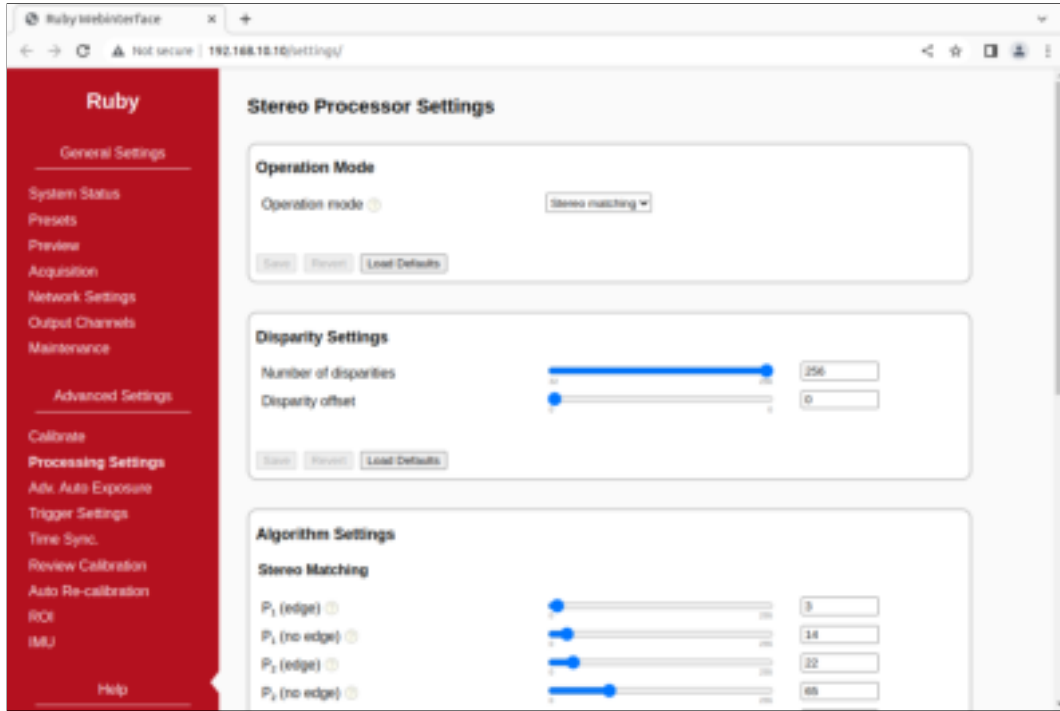


Figure 22: Screenshot of configuration page for processing settings.

9.9.2 Disparity Settings

If the operation mode is set to stereo matching, then the ‘*disparity settings*’ allow for a configuration of the disparity range that is searched by Ruby. The disparity range affects the frame rate that can be achieved. **The frame rate should be adjusted once the disparity range has been changed** (see Section 3.3 on page 5 for recommendations). Please be aware that increasing the disparity range will also reduce the maximum image size that can be configured.

The ‘*number of disparities*’ option specifies the total number of pixels that are searched for correspondences. This option has a high impact on the depth resolution and the covered measurement range (see Section 7.2). The start of the disparity range can be chosen through the ‘*disparity offset*’ option. Typically, a value of 0 is desired for the offset, which allows for range measurements up to infinity. If the observable distance is certain to be constrained, then low disparity values won’t occur. In this case it is possible to increase the disparity offset, such that these low disparities are not computed.

9.9.3 Algorithm Settings

The behavior of the image processing algorithms can be controlled through the ‘*algorithm settings*’. The default configuration has been determined using machine learning methods, and it should thus be the best choice for most use

cases. Nevertheless, all algorithm parameters can be adjusted through the web interface. The following parameters control the stereo matching algorithm:

Penalty for disparity changes (P_1): A penalty that is applied to gradually changing disparities. A large value causes gradual disparity changes to occur less frequently, while a small value causes gradual changes to occur more frequently. Different values can be configured for pixels that are on image edges (P_{1-edge}) and pixels that are not on edges ($P_{1-no-edge}$). These values must be smaller than the values for P_2 .

Penalty for disparity discontinuities (P_2): A penalty that is applied to abruptly changing disparities. A large value causes disparity discontinuities to occur less frequently, while a small value causes discontinuities to occur more frequently. Different values can be configured for pixels that are on image edges (P_{2-edge}) and pixels that are not on edges ($P_{2-no-edge}$). These values must be greater than the values for P_1 .

Ruby applies an optimization algorithm to improve the accuracy of the computed disparity map to sub-pixel resolution. If only a small region of interest (ROI) of the input image / disparity map is relevant, then this auto-tuning process can be constrained to only this ROI. In this case one should expect more accurate sub-pixel measurements inside the ROI. The relevant parameters for constraining the sub-pixel tuning ROI are:

Tune sub-pixel optimization on ROI: If enabled, the sub-pixel optimization is tuned on the region defined by the subsequent parameters, instead of the whole image.

Width: Width in pixels of the selected Region of Interest (ROI).

Height: Height in pixels of the selected ROI.

Offset X: Horizontal offset of the ROI relative to the image center.

Offset Y: Vertical offset of the ROI relative to the image center.

Ruby implements several methods for post-processing the computed disparity map. Each post-processing method can be activated or deactivated individually. The available methods are:

Mask border pixels: If enabled, this option marks all disparities that are close to the border of the visible image area as invalid, as they have a high uncertainty. This also includes all pixels for which no actual image data is available, due to the warping applied by the image rectification (see Section 7.1).

Consistency check: If enabled, stereo matching is performed in both matching directions, left-to-right and right-to-left. Pixels for which the disparity is not consistent are marked as invalid. The sensitivity of the consistency check can be controlled through the ‘*consistency check sensitivity*’ slider.

Uniqueness check: If enabled, pixels in the disparity map are marked as invalid if there is no sufficiently unique solution (i.e. the cost function does not have a global minimum that is significantly lower than all other local minima). The sensitivity of the uniqueness check can be controlled through the ‘*uniqueness check sensitivity*’ slider.

Texture filter: If enabled, pixels that belong to image regions with little texture are marked as invalid in the disparity map, as there is a high likelihood that these pixels are mismatched. The sensitivity of this filter can be adjusted through the ‘*texture filter sensitivity*’ slider.

Gap interpolation: If enabled, small patches of invalid disparities, which are caused by one of the preceding filters, are filled through interpolation.

Noise reduction: If enabled, an image filter is applied to the disparity map, which reduces noise and removes outliers.

Speckle filter iterations: Marks small isolated patches of similar disparity as invalid. Such *speckles* are often the result of erroneous matches. The number of iterations specify how aggressive the filter will be with removing speckles. A value of 0 disables the filter.

9.10 Advanced Auto Exposure and Gain Settings

To ensure the best possible image quality, Ruby provides a fully automatic exposure time and gain adaptation for rapidly changing lighting conditions, which often occurs in outdoor scenarios. You can activate and deactivate both auto functions independently on the *auto exposure* page, which is shown in Figure 23.

9.10.1 Exposure and Gain

Mode: Selects whether exposure time and/or gain are adjusted automatically. Under normal circumstances ‘*auto exposure and gain*’ should be selected for the automatic adjustment of both parameters.

Target intensity: Selects an average intensity value for the stereo images, which is targeted by the automatic adjustment. Intensity values are written in percentage numbers with 0 representing black and 100 white. Different values can be given for the color and monochrome sensors.

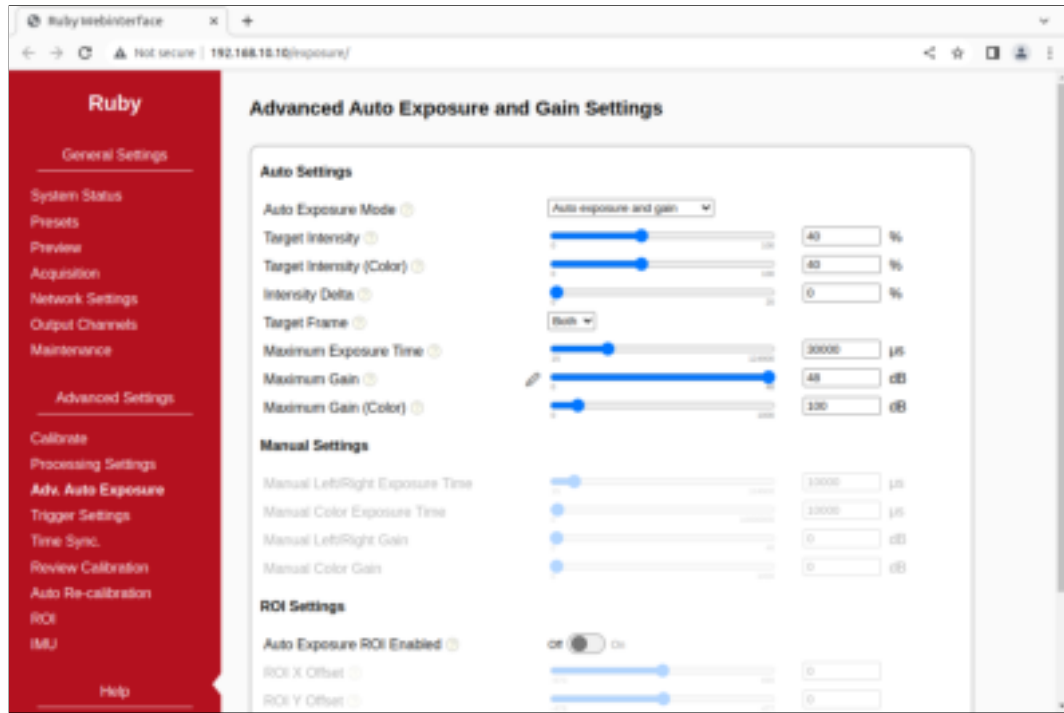


Figure 23: Screenshot of the configuration page for the automatic exposure and gain adjustment settings.

Target frame: Selects if the intensity of the *left* frame, the intensity of the *right* frame or the average intensity of *both* frames should be adjusted to the *target intensity*.

Maximum exposure time: A maximum value for the exposure time can be specified in order to limit motion blur. The value for the *maximum exposure time* should always be smaller than the time between two frames. Different values can be given for the color and monochrome sensors.

Maximum gain: Just like for the exposure time, it is also possible to constrain the maximum allowed gain. Constraining the gain can improve image processing results for situations with high sensor noise. Different values can be given for the color and monochrome sensors.

9.10.2 Manual Settings

If the automatic adjustment is deactivated in the *mode* selection, the exposure time and/or gain can be manually set to fixed values in this section.

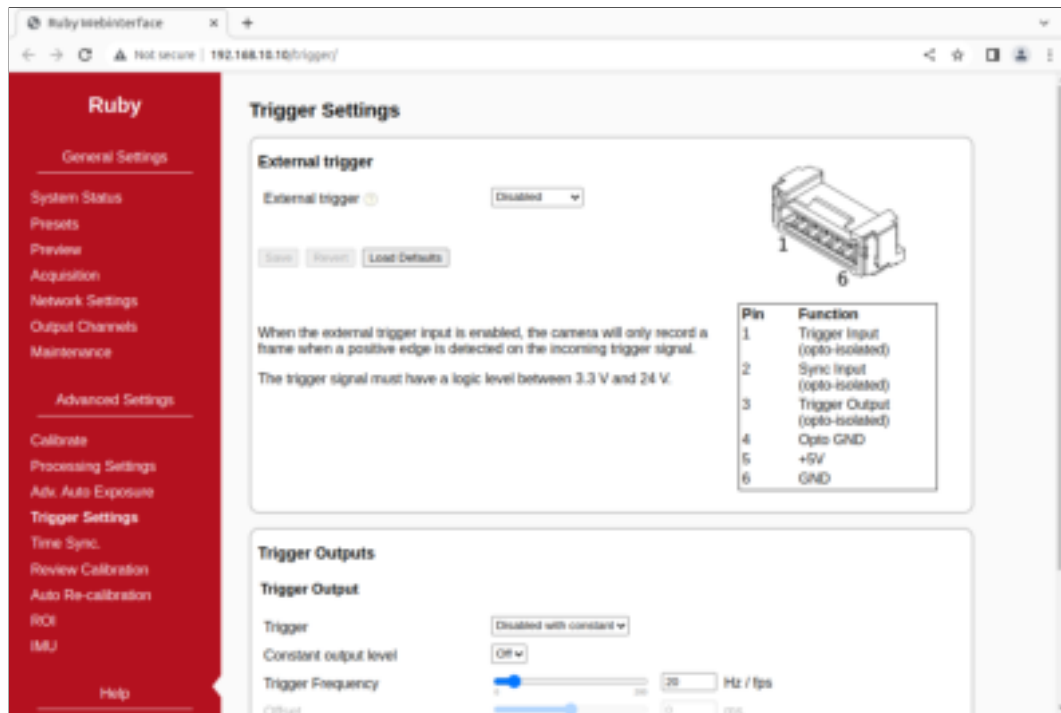


Figure 24: Screenshot of configuration page for trigger settings.

9.10.3 ROI Settings

Rather than performing the adjustment with respect to the average intensity of the complete image, you can compute the average intensity only on a region of interest. Enable ‘*use ROI for adjustment*’ in that case. ‘*Offset X*’ and ‘*Offset Y*’ describe the region’s center position relative to the image center. ‘*Width ROI*’ and ‘*Height ROI*’ let you adjust the spatial extension of the ROI. The ROI must be completely contained in the image. If this is not the case, the ROI will be cropped automatically.

9.11 Trigger Settings

The ‘*trigger settings*’ page that is shown in Figure 24 allows for a configuration of the trigger input and output. Ruby features a GPIO port that provides access to one trigger output and one trigger input signal. For electrical specifications of these signals please refer to Section 6.3.

When the trigger input is enabled, Ruby will only capture a frame when a signal pulse arrives at the trigger input pin, or if a software trigger is emitted through the API. For the hardware trigger signal, the exposure of the image sensor is started with the leading edge of the incoming signal. When the trigger input is enabled, the trigger output is not available.

When the trigger output is not enabled, it can be specified whether the

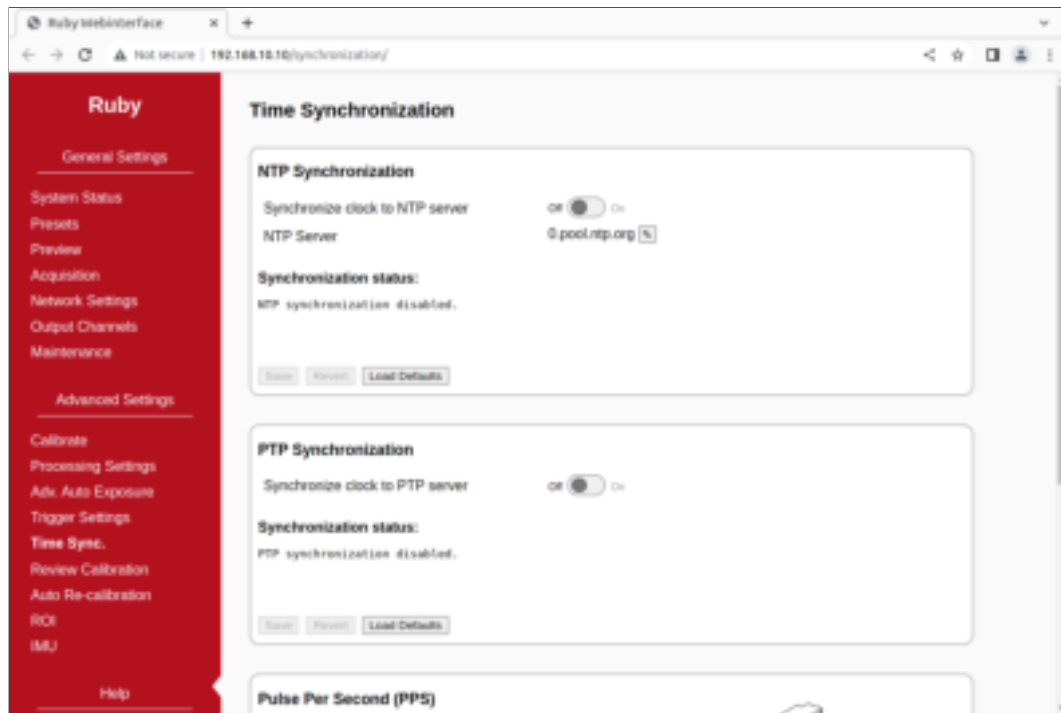


Figure 25: Screenshot of configuration page for time synchronization.

output should be tied to a *constant on* (logical 1) or *constant off* (logical 0). If enabled, the polarity of the generated signal can be either active-high or active-low. The pulse width can be constant or cycle between a list of pre-configured values.

The frequency of the trigger output will always match Ruby’s current frame rate. However, it is possible to specify a time offset, which is the delay from the start of sensor exposure to the the leading edge of the trigger output.

9.12 Time Synchronization

The ‘*time synchronization*’ page, which is shown in Figure 25, can be used to configure three possible methods for synchronizing Ruby’s internal clock. As explained in Section 7.4, the internal clock is used for timestamping captured frames.

The first option is to synchronize with a time server, using the Network Time Protocol (NTP) up to version 4. In this case Ruby synchronizes its internal clock to the given time server, using Coordinated Universal Time (UTC). The accuracy of the time synchronization depends on the latency of your network and time server. If NTP time synchronization is active, synchronization statistics are displayed in a dedicated status area.

As an alternative to NTP, the Precision Time Protocol (PTP) can be used for synchronization. PTP provides a significantly higher accuracy when com-

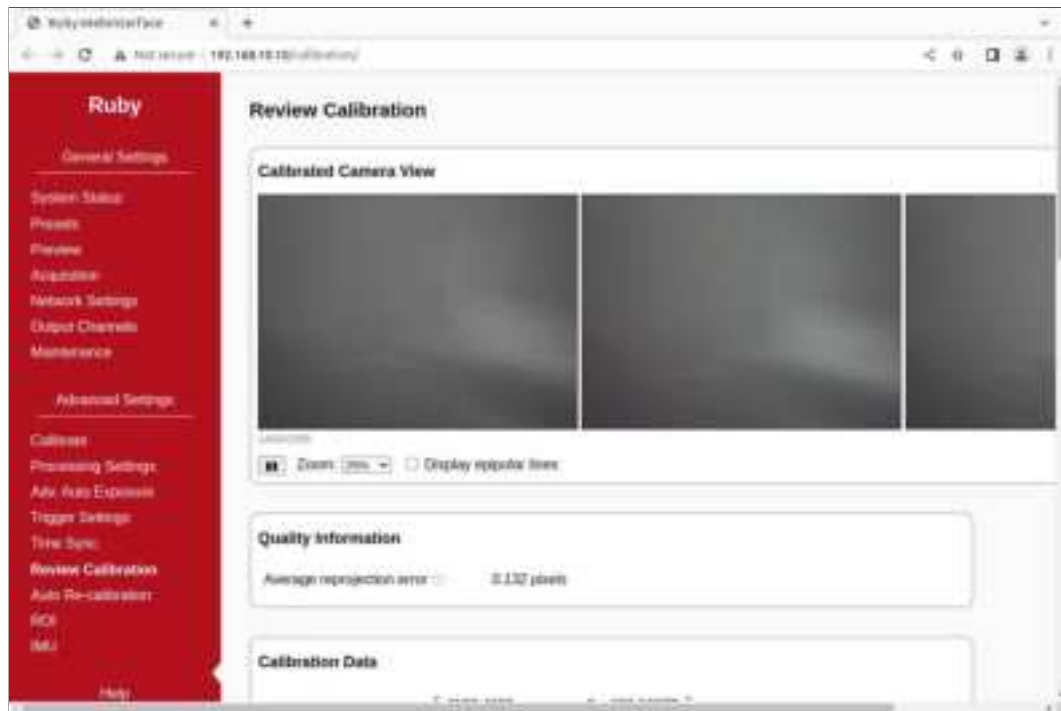


Figure 26: Screenshot of configuration page for reviewing camera calibration.

pared to NTP, and should hence be preferred if available. Like for NTP, the clock will also be set to UTC and synchronization status information will be displayed.

When using the Pulse Per Second (PPS) signal, the internal clock can be reset to 0 whenever a synchronization signal is received. Alternatively, the the system time stamp for the last received PPS signal can be transmitted with a captured frame. Please refer to Section 6.3.3 on page 11 for details on the PPS synchronization.

9.13 Reviewing Calibration Results

Once calibration has been performed, you can inspect the calibration results on the ‘*review calibration*’ page, which is shown in Figure 26. On the top of this page you can see a live preview of all image sensors as they are rectified with the current calibration parameters. Please make sure that corresponding points in the images of all image sensors have an identical vertical coordinate.

By activating the ‘*display epipolar lines*’ option, you can overlay a set of horizontal lines on the images. This allows for an easy evaluation of whether the equal vertical coordinates criterion is met. An example for a left and right input image with overlaid epipolar lines is shown in Figure 27.

In the ‘*quality information*’ section you can find the *average reprojection error*. This is a measure for the quality of your calibration, with lower val-



Figure 27: Example for evaluating vertical image coordinates.

ues indicating better calibration results. Please make sure that the average reprojection error is well below 1 pixel.

All computed calibration parameters are displayed in the ‘*calibration data*’ section. These parameters are:

M_1 , M_2 and M_3 : camera matrices for the left, right and color camera.

D_1 , D_2 and D_3 : distortion coefficients for the left, right and color camera.

R_1 , R_2 and R_3 : rotation matrices for the rotations between the original and rectified camera images.

P_1 , P_2 and P_3 : projection matrices in the new (rectified) coordinate systems.

Q_{12} : the disparity-to-depth mapping matrix for the left camera. See Section 7.2 for its use.

Q_{13} : the disparity-to-depth mapping matrix for the color camera (typically not needed).

T_{12} , T_{13} : translation vector between the coordinate systems of the left and right, and left and color cameras.

R_{12} , R_{13} : rotation matrix between the coordinate systems of the left and right, and left and color cameras.

The camera matrices M_i are structured as follows:

$$M_i = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad (1)$$

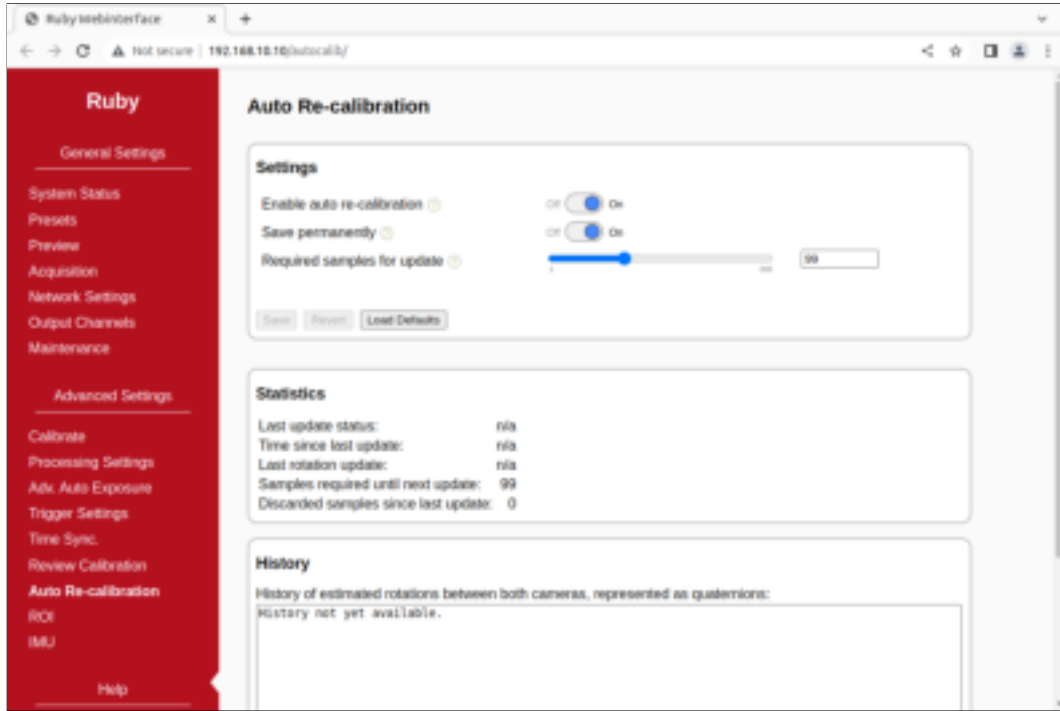


Figure 28: Screenshot of auto re-calibration settings.

where f_x and f_y are the lenses' focal lengths in horizontal and vertical direction (measured in pixels), and c_x and c_y are the image coordinates of the projection center.

The distortion coefficient vectors D_1 and D_2 have the following structure:

$$D_i = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix}, \quad (2)$$

where k_1 , k_2 and k_3 are radial distortion coefficients, and p_1 and p_2 are tangential distortion coefficients.

You can download all calibration information as a machine-readable YAML file, by clicking the download link at the bottom of the '*calibration data*' section. This allows you to easily import the calibration data into your own applications. Furthermore, you can save the calibration data to your PC and reload it at a later time, by using the '*upload calibration data*' section.

9.14 Auto Re-calibration

On the '*auto re-calibration*' page, which is shown in Figure 28, you can enable an automated estimation of the calibration parameters. In this case, the system remains calibrated even if the optical alignment is subject to variations.

Calibration parameters are usually divided into *intrinsic* parameters (focal length, projection center and distortion coefficients) and *extrinsic* parameters (transformation between the poses of all cameras). Auto re-calibration only

performs an update of the extrinsic parameters, as they are significantly more prone to variations. More specifically, only the rotation between the cameras is estimated. This is usually the most fragile parameter, which can be affected significantly by even minor deformations.

Auto re-calibration can be activated by selecting the ‘*enable auto re-calibration*’ option. Ruby will then continuously compute samples for the estimated inter-camera rotation. A robust estimation method is applied for selecting a final rotation estimate from the set of rotation samples. The *number of samples* that are used for this estimation process can be configured. Small sample sizes allow for a quick reaction on alignment variations, while large sample sizes allow for very accurate estimates. If the ‘*permanently save corrected calibration*’ option is selected, then the updated calibration is written to non-volatile memory and remains present even after a power cycle.

For the auto-calibration to work, the cameras must observe a scene with sufficient visual information. Ruby will identify salient image features and match them across all images. If not enough features can be detected, then auto re-calibration will not run. A typical scene should be sufficient for automatically re-calibrating the left and right monochrome cameras. For running the auto re-calibration on the color camera, however, a feature-rich black/white pattern is recommended. A white page with printed text, for example, serves well for this purpose.

In the *statistics* area you can find various information on the current performance of the auto calibration process. This includes the status of the latest re-calibration attempt, the time since the last calibration update, the rotational offset of the last update and the number of rotation samples that have been collected and discarded since the last update. Finally, you can find a list of recently computed inter-camera rotations in the *history* area. The listed rotations are represented as rotation quaternions.

9.15 Region of Interest

If not the entire sensor image is needed but only a subsection, then this can be configured on the ‘*region of interest*’ (ROI) page. This page will open a preview of the left and right images with overlaid frames showing the cropped region, which can be moved and resized in unison using the mouse (see Fig. 29). The device will revise the requested ROI dimensions; in this case you will see the region automatically snap to the closest valid image size.

If calibration was performed on a constrained centered window instead of the full sensor resolution (see Section 9.8), these constrained extents cannot be exceeded during ROI selection. The preview image size on the ROI selection page will reflect the constrained calibration-time resolution.

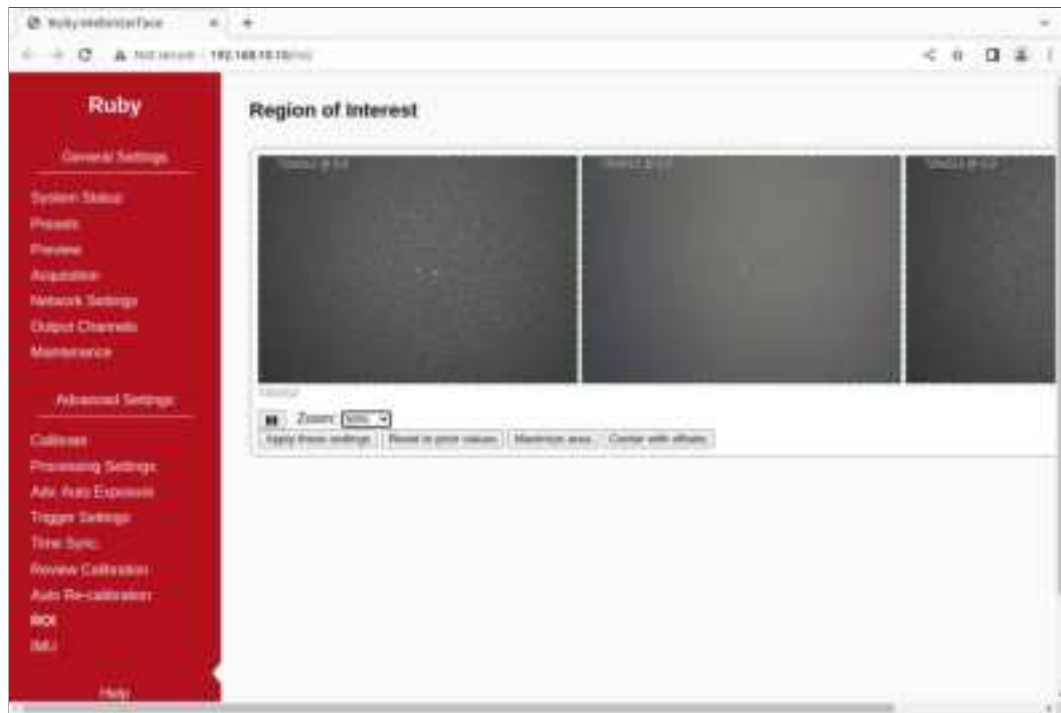


Figure 29: Screenshot of Region-of-Interest selection.

9.16 Inertial Measurement Unit

The inertial measurement unit (IMU) embedded in Ruby, which can provide real-time three-dimensional measurements for accelerometer, gyroscope, linear acceleration and magnetometer data, as well as integrated quaternion orientation readings, can be configured on the ‘*inertial measurement unit*’ page, which is shown in Figure 30.

In the ‘*network packet frequency*’ section, you can set the packet rate per second for the sensor readings. The value can be increased for minimum-latency (real-time) uses, or reduced for pure recording of time series, in which case longer data batches will be aggregated for each packet.

The sampling frequencies for the individual sensors can be configured in the ‘*sampling frequencies*’ section. Values range between 0 Hz (which disables a specific channel) and the maximum rate, which is 100 Hz for magnetometer data and 400 Hz for the other channels. The ‘*rotation quaternion*’ channel, which reflects the device orientation integrated from individual sensor channels, has an additional mode toggle: in ‘*absolute (geomagnetic)*’ mode, the device integrates the magnetometer to provide readings for the yaw angle (i.e. rotation around the gravity axis), thus estimating the absolute compass bearing. In ‘*relative (non-geomagnetic)*’ mode, no magnetometer data is used, and the yaw reading is based solely on motion integration – this entails a start at zero yaw, whatever the initial device orientation, and a progressively divergent

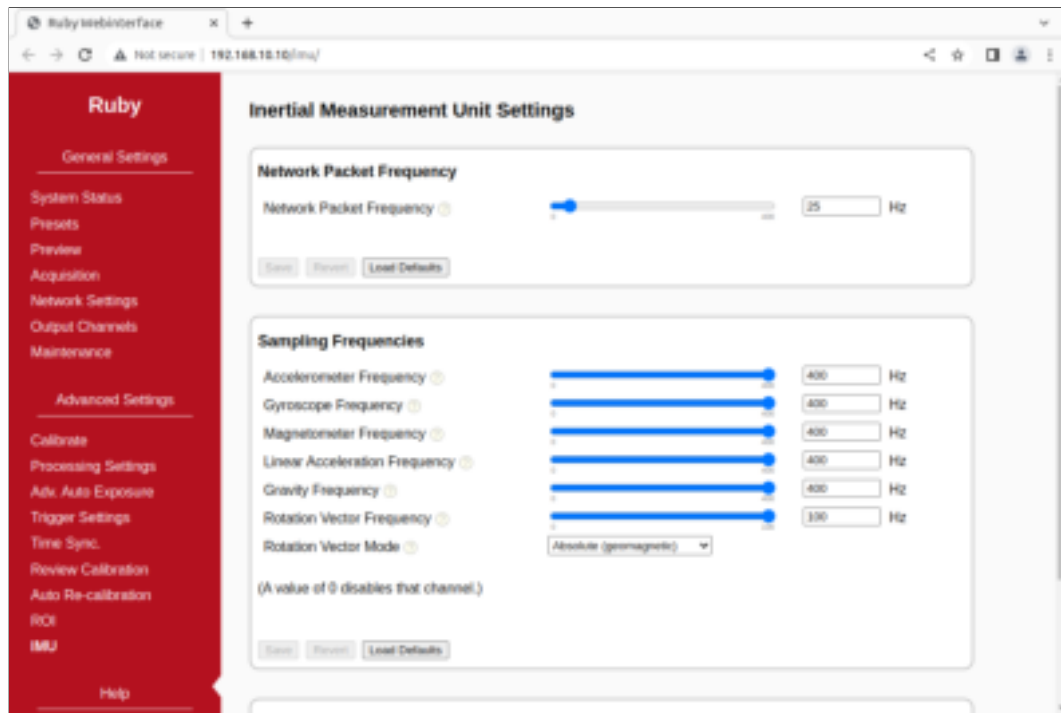


Figure 30: Screenshot of Inertial Measurement Unit Settings page.

drift of the reported yaw angle with respect to absolute compass directions.

9.16.1 Calibration of the inertial measurement unit

A live view of the orientation readings can be observed in the ‘*calibration / device orientation*’ section. Aside from the roll, pitch, and yaw angles, the calibration quality is reported on a scale from zero to three (reflecting the BNO08X levels of Unreliable; Low Accuracy; Medium Accuracy; and High Accuracy). The estimated accuracy for the yaw (compass bearing) angle is reported if ‘absolute (geomagnetic)’ mode is enabled. The magnetometer readings are the least reliable component, therefore the calibration status may be reported as less accurate in ‘absolute (geomagnetic)’ mode.

The ‘*start calibration*’ button puts the IMU in calibration mode. The recommended procedure is to then orient the device into five to six perpendicular directions (corresponding to cube faces) with different rotations, and briefly keeping the device still at each of those orientations. The calibration status should progressively improve to level 2 or 3. The ‘*finish calibration*’ button saves the new calibration data and resets the IMU – readings will resume after a short moment with the new base calibration, which will then persist through power cycles.

10 API Usage Information

10.1 General Information

The cross-platform *libvisiontransfer* C++ and Python API is available for interfacing custom software with Ruby. For Windows, a binary version of the library is available that can be used with Microsoft Visual Studio. For Linux, please compile the library from the available source code. The API is included as part of the available software release, which can be downloaded from our support website¹.

The *libvisiontransfer* API provides functionality for receiving the processing results of Ruby over a computer network. Furthermore, the API also allows for the transmission of image data. It can thus be used for emulating Ruby when performing systems development.

The transmitted processing results consist of a set of images. Usually these are the rectified left image and the computed disparity map. If configured, however, Ruby can also provide the raw recorded images or all rectified images (see Section 9.9).

Original and rectified camera images are typically transmitted with a monochrome bit-depth of 8 bits or 12 bits per pixel, or in 8-bit RGB mode. The disparity map is always transmitted with a bit depth of 12 bits. Inside the library, the disparity map and any 12-bit images are inflated to 16 bits, to allow for more efficient processing.

The API provides three classes that can be used for receiving and transmitting image data:

- **ImageProtocol** is the most low-level interface. This class allows for the encoding and decoding of image sets to / from network messages. You will have to handle all network communication yourself.
- **ImageTransfer** opens up a network socket for sending and receiving image sets. This class is single-threaded and will thus block when receiving or transmitting data.
- **AsyncTransfer** allows for the asynchronous reception or transmission of image sets. This class creates one or more threads that handle all network communication.

Detailed information on the usage of each class can be found in the available API documentation.

10.2 ImageTransfer Example

An example for using the class **ImageTransfer** in C++ to receive processing results over the network, and writing them to image files, is shown below.

¹<https://nerian.com/support/software/>

This source code file is part of the API source code release. Please refer to the API documentation for further information on using ImageTransfer and for examples in Python.

```
#include <visiontransfer/deviceenumeration.h>
#include <visiontransfer/imagetransfer.h>
#include <visiontransfer/imageset.h>
#include <iostream>
#include <exception>
#include <stdio.h>

#ifdef _MSC_VER
// Visual studio does not come with snprintf
#define snprintf _snprintf_s
#endif

using namespace visiontransfer;

int main() {
    // Search for Nerian stereo devices
    DeviceEnumeration deviceEnum;
    DeviceEnumeration::DeviceList devices =
        deviceEnum.discoverDevices();
    if(devices.size() == 0) {
        std::cout << "No devices discovered!" << std::endl;
        return -1;
    }

    // Print devices
    std::cout << "Discovered devices:" << std::endl;
    for(unsigned int i = 0; i < devices.size(); i++) {
        std::cout << devices[i].toString() << std::endl;
    }
    std::cout << std::endl;

    // Create an image transfer object that receives data from
    // the first detected device
    ImageTransfer imageTransfer(devices[0]);

    // Receive 100 images
    for(int imgNum=0; imgNum<100; imgNum++) {
        std::cout << "Receiving image set" << imgNum << std::endl;

        // Receive image
        ImageSet imageSet;
        while(!imageTransfer.receiveImageSet(imageSet)) {
            // Keep on trying until reception is successful
        }

        // Write all included images one after another
        for(int i = 0; i < imageSet.getNumberOfImages(); i++) {
            // Create PGM file

```

```

        char fileName[100];
        snprintf(fileName, sizeof(fileName), "image%03d_%d.pgm", i,
            imgNum);

        imageSet.writePgmFile(i, fileName);
    }
}

return 0;
}

```

10.3 AsyncTransfer Example

An example for using the class `AsyncTransfer` in C++ to receive processing results over the network, and writing them to image files, is shown below. This source code file is part of the API source code release. Please refer to the API documentation for further information on using `AsyncTransfer` and for examples in Python.

```

#include <visiontransfer/deviceenumeration.h>
#include <visiontransfer/asynctransfer.h>
#include <visiontransfer/imageset.h>
#include <iostream>
#include <exception>
#include <stdio.h>

#ifdef _MSC_VER
// Visual studio does not come with snprintf
#define snprintf _snprintf_s
#endif

using namespace visiontransfer;

int main() {
    try {
        // Search for Nerian stereo devices
        DeviceEnumeration deviceEnum;
        DeviceEnumeration::DeviceList devices =
            deviceEnum.discoverDevices();
        if(devices.size() == 0) {
            std::cout << "No devices discovered!" << std::endl;
            return -1;
        }

        // Print devices
        std::cout << "Discovered devices:" << std::endl;
        for(unsigned int i = 0; i < devices.size(); i++) {
            std::cout << devices[i].toString() << std::endl;
        }
        std::cout << std::endl;
    }
}

```

```

// Create an image transfer object that receives data from
// the first detected device
AsyncTransfer asyncTransfer(devices[0]);

// Receive 100 images
for(int imgNum=0; imgNum<100; imgNum++) {
    std::cout << "Receiving image set" << imgNum << std::endl;

    // Receive image
    ImageSet imageSet;
    while(!asyncTransfer.collectReceivedImageSet(imageSet,
        0.1 /*timeout*/)) {
        // Keep on trying until reception is successful
    }

    // Write all included images one after another
    for(int i = 0; i < imageSet.getNumberOfImages(); i++) {
        // Create PGM file
        char fileName[100];
        snprintf(fileName, sizeof(fileName), "image%03d_%d.pgm", i,
            imgNum);

        imageSet.writePgmFile(i, fileName);
    }
}
} catch(const std::exception& ex) {
    std::cerr << "Exception occurred:" << ex.what() << std::endl;
}

return 0;
}

```

10.4 3D Reconstruction

As described in Section 7.2, the disparity map can be transformed into a set of 3D points. This requires knowledge of the disparity-to-depth mapping matrix Q (see Section 7.2), which is transmitted by Ruby along with each disparity map.

An optimized implementation of the required transformation, which uses the SSE or AVX instruction sets, is provided by the API through the class `Reconstruct3D`. This class converts a disparity map into a map of 3D point coordinates. Please see the API documentation for further details.

10.5 Parameters

A separate network protocol is used for reading and writing device parameters. This protocol is implemented by `DeviceParameters`. Any parameters that are changed through this protocol will be reset if the device is rebooted or if the user makes a parameter change through the web interface.

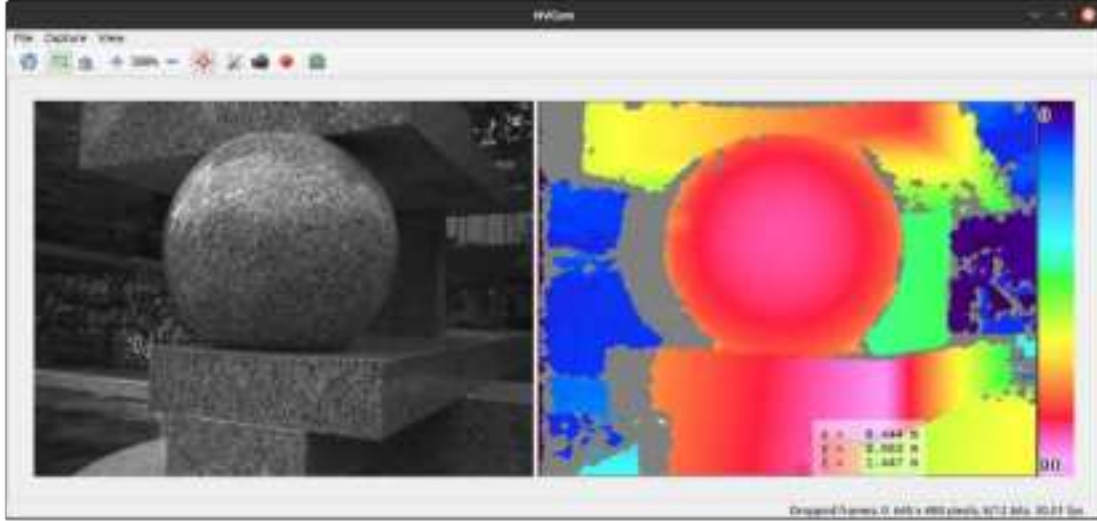


Figure 31: Screenshot of NVCom application.

11 Supplied Software

11.1 NVCom

The available source code or binary software release also includes the NVCom client application, which is shown in Figure 31. When compiling this application yourself, please make sure that you have the libraries OpenCV and Qt installed. NVCom provides the following features:

- Discover Ruby devices, view their status, and access their setup.
- Receive and display images and disparity maps from Ruby.
- Perform color-coding of disparity maps.
- Provide live 3D pointcloud visualization.
- Write received data to files as images or 3D point clouds.

NVCom comes with a GUI that provides access to all important functions. More advanced features are available through the command line options, which are listed in Table 2. The command line options can also be used for automating data recording or playback.

Unless NVCom is run in non-graphical mode, it opens a GUI window that displays the received images. The currently displayed image set can be written to disk by pressing the *enter* key or by clicking the camera icon in the toolbar. When pressing the *space* key or clicking the recording icon, all subsequent images will be saved. When closing NVCom it will save its current settings, which will be automatically re-loaded when NVCom is launched the next time.

Table 2: Available command line options for NvCom.

-c <i>VAL</i>	Select color coding scheme (0 = no color, 1 = red / blue, 2 = rainbow)
-f <i>FPS</i>	Limit send frame rate to FPS
-w <i>DIR</i>	Immediately write all images to DIR
-s <i>DIR</i>	Send the images from the given directory
-n Non-graphical	
-p <i>PORT</i>	Use the given remote port number for communication
-H <i>HOST</i>	Use the given remote hostname for communication
-t <i>on/off</i>	Activate / deactivate TCP transfers
-d	Disable image reception
-T	Print frame timestamps
-3 <i>VAL</i>	Write a 3D point cloud with distances up to VAL (0 = off)
-z <i>VAL</i>	Set zoom factor to VAL percent
-F	Run in fullscreen mode
-b <i>on/off</i>	Write point clouds in binary rather than text format
-h, -help	Displays this help.

11.2 GenICam GenTL Producer

11.2.1 Installation

The available software release further includes a software module that complies to the GenICam GenTL standard. The GenTL standard specifies a generic transport layer interface for accessing cameras and other imaging devices. According to the GenICam naming convention, a GenTL producer is a software driver that provides access to an imaging device through the GenTL interface. A GenTL consumer, on the other hand, is any software that uses one or more GenTL producers through this interface. The supplied software module represents a GenTL producer and can be used with any application software that acts as a consumer. This allows for the ready integration of Ruby into existing machine vision software suites like e.g. HALCON.

Depending on the version that you downloaded, the producer is provided either as a binary or as source code. If you choose the source code release, the producer will be built along with the other software components. The produced / downloaded binary is named **nerian-gentl.cti**. In order to be found by a consumer, this file has to be placed in a directory that is in the GenTL search path. The search path is specified through the following two environment variables:

GENICAM_GENTL32_PATH: Search path for 32-bit GenTL producers.
GENICAM_GENTL64_PATH: Search path for 64-bit GenTL producers.

The binary Windows installer automatically configures these environment

variables. When building the source code release, please configure the environment variables manually.

11.2.2 Virtual Devices

Once the search path has been set, the producer is ready to be used by a consumer. For each Ruby the producer provides five *virtual devices*, which each deliver one part of the obtained data. These virtual devices are named as follows:

- /color** Provides the color camera image that is transmitted by Ruby. In the default configuration, this is the image after rectification and projection has been applied. The image is encoded as an RGB image with 8 bits per channel (RGB8).
- /left** Provides the left camera image that is transmitted by Ruby. In the default configuration, this data stream is not available. The image is encoded with 8 or 12 bits per pixel (Mono8 or Mono12).
- /right** Provides the right camera image. In the default configuration, this data stream is not available. The image is encoded in Mono8 or Mono12 format.
- /disparity** Provides the disparity map that is transmitted by Ruby. This data is not available if Ruby is configured in pass through or rectify mode. The disparity map is transmitted with a non-packed 12 bits per pixel encoding (Mono12).
- /pointcloud** Provides a transformation of the disparity map into a 3D point cloud (see Section 7.2). Each point is represented by three 32-bit floating point numbers that encode an x-, y- and z-coordinate (Coord3D_ABC32f).
- /** This virtual device provides a multi-part data stream which contains all the data that is available through the other devices. In the default configuration, this device provides the left camera image, the disparity map and the 3D point cloud.

The virtual devices */color*, */left*, */right* and */disparity* deliver the unprocessed data that is received from Ruby. The data obtained through the */pointcloud* device is computed by the producer from the received disparity map. This is done by multiplying the disparity map with the disparity-to-depth mapping matrix Q (see Section 7.2), which is transmitted by Ruby along with each image pair. Invalid disparities are set to the minimum disparity and thus result in points with very large distances.

It is recommended to use the multi-part virtual device */* when more than one type of data is required. This will guarantee that all data acquisition is synchronized. When requiring only one type of input data, then using the dedicated virtual devices is the most efficient option.

11.2.3 Device IDs

All device IDs that are assigned by the producer are URLs and consist of the following components:

```
protocol://address/virtual device
```

The *protocol* component identifies the underlying transport protocol that shall be used for communication. The following values are possible:

udp: Use the connection-less UDP transport protocol for communication.

tcp: Use the connection oriented TCP transport protocol for communication.

The *virtual device* shall be set to one of the device names that have been listed in the previous section. Some examples for valid device IDs are:

```
udp://192.168.10.10/pointcloud
tcp://192.168.10.100/left
```

11.3 ROS Node

For integrating Ruby with the Robot Operating System (ROS), there exists an official ROS node. This node is called `nerian_stereo` and can be found in the official ROS package repository. The node publishes the computed disparity map and the corresponding 3D point cloud as ROS topics. Furthermore, it can publish camera calibration information and IMU readings.

To install this node from the ROS package servers on a Ubuntu Linux system, please use the following commands:

```
> sudo apt-get update
> sudo apt-get install ros-`rosversion -d`-nerian-stereo
```

Detailed information on this node can be found on the corresponding ROS wiki page².

12 Support

If you require support with using Ruby then please use our support forum at <https://nerian.com/support/forum/> or contact:

Nerian Vision GmbH
Zettachring 2
70567 Stuttgart
Germany

²http://wiki.ros.org/nerian_stereo

Phone: +49 711 2195 9414

E-mail: service@nerian.com

Website: www.nerian.com

13 Warranty Information

The device is provided with a 2-year warranty according to German federal law (BGB). Warranty is lost if:

- the housing is opened by others than official Nerian Vision Technologies service staff.
- the firmware is modified or replaced, except for official firmware updates.

In case of warranty please contact our support staff.

14 Open Source Information

Ruby's firmware contains code from the open source libraries and applications listed in Table 3. Source code for these software components and the wording of the respective software licenses can be obtained from the open source information website³. Some of these components may contain code from other open source projects, which may not be listed here. For a definitive list, please consult the respective source packages.

The following organizations and individuals have contributed to the various open source components:

Free Software Foundation Inc., Emmanuel Pacaud, EMVA and contributors, The Android Open Source Project, Red Hat Incorporated, University of California, Berkeley, David M. Gay, Christopher G. Demetriou, Royal Institute of Technology, Alexey Zelkin, Andrey A. Chernov, FreeBSD, S.L. Moshier, Citrus Project, Todd C. Miller, DJ Delorie, Intel Corporation, Henry Spencer, Mike Barcroft, Konstantin Chuguev, Artem Bityuckiy, IBM, Sony, Toshiba, Alex Tatmanjants, M. Warner Losh, Andrey A. Chernov, Daniel Eischen, Jon Beniston, ARM Ltd, CodeSourcery Inc, MIPS Technologies Inc, Intel Corporation, Willow Garage Inc., NVIDIA Corporation, Advanced Micro Devices Inc., OpenCV Foundation, Itseez Inc., The Independent JPEG Group, elibThomas G. Lane, Guido Vollbeding, Simon-Pierre Cadieux, Eric S. Raymond, Mans Rullgard, Cosmin Truta, Gilles Vollant, James Yu, Tom Lane, Glenn Randers-Pehrson, Willem van Schaik, John Bowler, Kevin Bracey, Sam Bushell, Magnus Holmgren, Greg Roelofs, Tom Tanner, Andreas Dilger, Dave Martindale, Guy Eric Schalnat, Paul Schmidt, Tim Wegner, Sam Leffler, Silicon Graphics, Inc. Industrial Light & Magic, University of Delaware, Martin Burnicki, Harlan Stenn, Danny Mayer, The PHP Group, OpenSSL Software Services, Inc., OpenSSL Software Foundation, Inc., Andy Polyakov, Ben Laurie, Ben Kaduk, Bernd Edlinger, Bodo Möller, David Benjamin, Emilia Käsper, Eric Young, Geoff Thorpe, Holger Reif, Kurt Roeckx, Lutz Jänicke, Mark J. Cox, Matt Caswell, Matthias St. Pierre, Nils Larsch, Paul Dale, Paul C. Sutton, Ralf S. Engelschall, Rich Salz, Richard Levitte, Stephen Henson, Steve Marquess, Tim Hudson, Ulf Möller, Viktor Dukhovni

³<http://nerian.com/support/resources/scenescan-open-source/>

All authors contributing to packages included in PetaLinux. Please obtain the full list from www.xilinx.com/petalinux.

If you believe that your name should be included in this list, then please let us know.

Table 3: Open source components.

Name	Version	License(s)
Aravis	0.6.4 patched	GNU LGPL 2.0
GenApi reference implementation	3.1.0	GenICam License
libgpiod	1.4	GNU LGPL 2.1
libwebsockets	2.2	GNU LGPL 2.1
Linux PTP	3.1	GNU GPL 2
ntp	4.2.8p10	BSD License
OpenCV	3.2.0	MIT License
		BSD License
		libpng License
OpenSSL	1.1.1d	JasPer License 2.0
		BSD License
PetaLinux	2019.2	Various
PHP	7.3.7	PHP License

Revision History

Revision	Date	Author(s)	Description
v1.0	September 28, 2022	KS	Initial version
v0.1	August 23, 2022	KS	Preliminary draft