Table of Contents

I have a stereo image pair. Can we stereo rectify it and perform some sort of triangulation or distance estimation to objects in images?

Understanding stereo rectification

Stereo rectification involves aligning images from two cameras to make them parallel and rectilinear, enabling easier depth estimation through disparity computation and triangulation.

Understanding stereo rectification

Stereo rectification involves computing intrinsic and extrinsic parameters, disparity maps, and leveraging OpenCV functions for calibration. Triangulation converts matched points to 3D coordinates using projection matrices and linear methods.

Mapping stereo depth

I'm calculating depth using geometric setups: coordinates in both images, baseline, and disparity. Precision hinges on calibration details, streamlining stereo matching by aligning corresponding points.

Focusing on stereo rectification

I'm reprojecting images to align epipolar lines horizontally, simplifying matching. Triangulation follows once corresponding points and camera geometry are established, with various algorithmic options available.

Utilizing OpenCV

Stereo rectification with OpenCV requires calibrated cameras or known intrinsic parameters. If calibration isn't available, epipolar geometry computation is an alternative, but accurate triangulation is best with known calibration.

Mapping stereo vision

First, calibrate the camera. Next, rectify images, compute disparity, and gauge depth. Finally, use OpenCV's triangulatePoints for 3D point mapping, ensuring precision throughout each step.

Yes, you can. The general approach involves two main stages:

1. **Stereo Rectification:**
   - **Purpose:** Rectification reprojects the images so that the epipolar lines become horizontal. This simplifies the search for corresponding points between the two images (they'll lie on the same row).
   - **How to Do It:**
     - **Calibration:** First, you typically need to calibrate your stereo camera system. Calibration gives you the intrinsic parameters (focal length, principal point, distortion coefficients) for each camera and the extrinsic parameters (rotation and translation between the cameras). If you don't have calibration data, you can estimate the epipolar geometry by computing the fundamental matrix, but full calibration is preferable for accurate depth estimates.
     - **Rectification Algorithms:** With calibration data, you can use functions such as OpenCV's `cv::stereoRectify` (or the Python equivalent) to compute rectification transforms. Once you have these, you use `cv::initUndistortRectifyMap` and `cv::remap` to actually warp the images into a rectified coordinate system.
2. **Triangulation and Distance Estimation:**
   - **Triangulation Basics:** Once you have rectified images, you find corresponding points (using feature matching, block matching, or semi-global matching algorithms) and compute the disparity (the horizontal shift between corresponding points).
   - **Depth Calculation:** With known camera parameters (baseline and focal length), the depth Z of a point can be computed using the relation:

$$Z = \frac{f \cdot B}{d}$$

where:

- f is the focal length,
- B is the baseline (distance between the two camera centers),
- d is the disparity (difference in x-coordinates of the corresponding points in the rectified images).

- **Triangulation Methods:** For more general 3D reconstruction, you can use triangulation. This involves using the projection matrices (obtained during calibration/rectification) to back-project the corresponding image points into 3D space. OpenCV provides the `cv::triangulatePoints` function for this purpose, which uses linear methods (often based on SVD) to compute the 3D point that best fits the two projection rays.

A Simple Example Using OpenCV (Python)

Here's a rough outline of what the process might look like:

python

```python
import cv2
import numpy as np

# Assume you have already obtained the following through calibration:
# cameraMatrix1, distCoeffs1 for the left camera,
# cameraMatrix2, distCoeffs2 for the right camera,
# R (rotation) and T (translation) between the cameras.

# Image sizes (width, height)
image_size = (width, height)

# Compute rectification transforms
R1, R2, P1, P2, Q, _, _ = cv2.stereoRectify(cameraMatrix1, distCoeffs1,
                                            cameraMatrix2, distCoeffs2,
                                            image_size, R, T)

# Compute the rectification maps for both cameras
map1x, map1y = cv2.initUndistortRectifyMap(cameraMatrix1, distCoeffs1, R1, P1, image_size,
cv2.CV_32FC1)
map2x, map2y = cv2.initUndistortRectifyMap(cameraMatrix2, distCoeffs2, R2, P2, image_size,
cv2.CV_32FC1)

# Load your stereo images
img_left = cv2.imread('left_image.png')
img_right = cv2.imread('right_image.png')

# Rectify the images
```

```
rect_left = cv2.remap(img_left, map1x, map1y, cv2.INTER_LINEAR)
rect_right = cv2.remap(img_right, map2x, map2y, cv2.INTER_LINEAR)

# Now, perform stereo matching to compute the disparity map.
# One common approach is using StereoSGBM:
min_disp = 0
num_disp = 16*6  # must be divisible by 16
stereo = cv2.StereoSGBM_create(minDisparity=min_disp,
                               numDisparities=num_disp,
                               blockSize=7,
                               P1=8*3*7**2,
                               P2=32*3*7**2,
                               disp12MaxDiff=1,
                               uniquenessRatio=10,
                               speckleWindowSize=100,
                               speckleRange=32)
disparity = stereo.compute(rect_left, rect_right).astype(np.float32) / 16.0

# Once you have the disparity, you can compute depth.
# For a given pixel, the depth is:
# depth = (focal_length * baseline) / disparity
# If you want a full 3D reconstruction, you can reproject the disparity image to 3D:
points_3D = cv2.reprojectImageTo3D(disparity, Q)

# Alternatively, if you have specific corresponding points (pt_left and pt_right), you can use:
points_left = np.array([[pt_left_x], [pt_left_y]], dtype=float)
points_right = np.array([[pt_right_x], [pt_right_y]], dtype=float)
# Use triangulation with projection matrices P1 and P2
point_4d_hom = cv2.triangulatePoints(P1, P2, points_left, points_right)
point_3d = point_4d_hom / point_4d_hom[3]  # convert from homogeneous to 3D
```

Key Considerations

- **Calibration Accuracy:** The quality of your distance estimation depends heavily on accurate calibration. Errors in the intrinsic/extrinsic parameters will propagate into the 3D reconstruction.
- **Matching Quality:** Good rectification makes matching easier, but the matching algorithm still needs to be robust, especially in textureless regions or areas with repetitive patterns.
- **Baseline and Disparity:** A larger baseline generally improves depth resolution for nearby objects, but too large a baseline can introduce occlusions. Also, very small disparities (for far-away objects) can be noisy.

Summary

Yes, after stereo rectifying your image pair (which aligns the images for easier correspondence matching), you can compute disparities and use triangulation (or the simple depth formula) to estimate the distance to objects in your scene. This process is widely used in stereo vision applications, from robotics to 3D reconstruction.

If you need more details or have a specific implementation question, feel free to ask!

Frame Size 1024 x 1024
Spectrum visible
quantization 8 bit grayscale
camera height 325 mm
camera cant angle (vertical) 11 degree towards ground
camera cant angle (horizontal) 0 degree
inter camera distance 240 mm
focal length 21.6 mm
detector size 15 micro x 15 micro
FOV 39 degree x 39 degree
Image size on sensor 1.5 mm x 8.5 mm

Mapping camera parameters

I'm seeing details like frame size, spectrum, and quantization, which likely aim to enable accurate distance calculation from stereo images. These could refine the process of triangulation or distance estimation.

Identifying the request

OK, I'm thinking through how these camera parameters might apply to tasks like computing extrinsics, projection matrices, stereo rectification, or depth estimation, though the exact request from the user is unclear.

Analyzing camera parameters

I'm examining details such as frame size, spectrum, quantization, camera height, cant angles, inter-camera distance, focal length, detector size, field of view, and image size on the sensor, noting a possible contradiction in sensor size.

Examining image dimensions

The dimensions "1.5 mm x 8.5 mm" likely represent the actual captured image area on the sensor, rather than the physical size, as they don't match the pixel count.

Calculating distance

I'm mapping out the distance estimation by converting the focal length into pixel units using the pixel size, which helps in stereo triangulation and accurate depth calculation.

Confirming sensor dimensions

I'm noting that the sensor width for a 39-degree field of view calculates to approximately 15.3 mm, matching earlier findings, while the specified "Image size on sensor" might represent captured image area.

Implementing triangulation

I'm gathering info on triangulation with focal length in pixel units and inter-camera distance, converting to pixel units, and using the formula Z = (f * B) / d to compute object distance from image disparity.

Calculating distance

I'm piecing together the distance by using the formula Z = (f_pixels * B) / disparity, after converting the focal length to pixel units, and considering baseline and disparity in pixels.

Calculating ground distance

I'm examining how to compute ground intersection coordinates using the camera height, tilt angle, and stereoscopic triangulation, ensuring these parameters are integrated into the calculation.

Calculating intrinsic matrix

I'm working out the intrinsic matrix K, which involves determining focal lengths in pixel units and principal points from provided camera parameters, advancing to stereo triangulation and projection matrix computation.

Aligning focal lengths

I'm checking the focal length accuracy by comparing pixel units with the sensor's field of view, confirming consistency despite slight differences in sensor size assumptions.

Adjusting the perspective

I'm assuming the sensor's image size is influenced by a region of interest, with the camera's 11-degree tilt adding rotation to the projection matrix.

Aligning cameras

OK, let me see. Assuming both cameras are similarly aligned, the baseline is 240 mm, and stereo rectification is used to counteract the 11-degree tilt during triangulation.

Formulating depth estimation

Using a 240 mm baseline and a 1440 px focal length, depth estimation applies the formula: depth (in mm) = (focal length in pixels * baseline) / disparity (in pixels).

Incorporating tilt effects

Taking into account the cameras' tilt of 11 degrees and the 39-degree field of view, relevant parameters are integrated into the stereo system for accurate distance estimation.

Mapping camera parameters

I'm converting focal length to pixels, detailing the camera matrix, and adjusting for the 11-degree tilt with the extrinsic matrix and baseline of 240 mm along the x-axis.

Aligning image parameters

I'm mapping out image coordinate conversions to metric coordinates, while addressing how the 11-degree tilt and baseline adjustments link X-axis translations with specific transformations.

Reconciling pixel pitch

I'm examining the image size on the sensor, considering the effective pixel pitch, and ensuring it's consistent with the provided parameters for calibration and stereo geometry computation.

Formulating the intrinsic matrix

I'm calculating the intrinsic matrix with an 11-degree tilt, adjusting rotational matrices to align with the downward optical axis, and approximating the principal point for the left camera.

Piecing together

I'm organizing projection matrices, calculating disparity maps, and refining depth estimation by utilizing the 11-degree tilt and baseline for more accurate stereo vision assessments.

Analyzing object positioning

By back-projecting pixel coordinates and considering the camera's tilt and height, I can accurately determine the object's distance from the ground plane using geometry.

Adjusting parameters

I'm tweaking effective pixel pitch calculations to double-check alignment with sensor dimensions, ensuring consistent and precise stereo pipeline implementation. Reflecting on potential inconsistencies and their impacts.

Adjusting pixel pitch

I'm verifying the effective pixel pitch to align with the 1.5 mm x 8.5 mm sensor dimensions, ensuring it aligns with the 1024 pixel count for accurate intrinsic and extrinsic matrix computations.

Aligning pixel pitch

OK, let me see. I'm ensuring the effective pixel pitch matches the sensor's image size calculations for precise stereo pipeline refinement, addressing discrepancies for accurate intrinsic and extrinsic matrices.

Understanding sensor size

I'm thinking through how the detector size might affect the sensor's effective area, influencing image calculations and requiring careful consideration for triangulation and depth estimation.

Below is an outline of how you can use your camera's parameters in a stereo setup to perform rectification and distance estimation. In a nutshell, you need to:

1. **Compute the Intrinsic Parameters**
2. **Incorporate the Extrinsic (Pose) Information**
3. **Rectify the Stereo Pair**
4. **Compute Disparity and then Depth**

I'll go through each step while showing how your given parameters come into play.

---

1. Compute the Intrinsic Parameters

Your camera parameters include:

- **Focal length:** 21.6 mm
- **Detector (pixel) size:** 15 µm × 15 µm
- **Image resolution:** 1024 × 1024 pixels

A key step is to convert the focal length into pixel units. Since 15 µm = 0.015 mm, the focal length in pixels is

$$f_{\text{px}} = \frac{21.6\,\text{mm}}{0.015\,\text{mm/px}} = 1440\,\text{px}.$$

If you assume that the principal point is at the center of the image (which is typical unless you have reason to choose otherwise), then

$$c_x = c_y = \frac{1024}{2} = 512\,\text{px}.$$

Thus, the intrinsic (camera) matrix K becomes:

$$K = \begin{bmatrix} 1440 & 0 & 512 \\ 0 & 1440 & 512 \\ 0 & 0 & 1 \end{bmatrix}.$$

Note: You mentioned an "Image size on sensor" of 1.5 mm × 8.5 mm, which is not consistent with a 1024 px array at 15 µm/pixel (which would span roughly 15.36 mm on a side). In practice you must be sure that your effective sensor size and pixel pitch are consistent. (It may be that only a sub-region of the full sensor is used.)

---

2. Incorporate the Extrinsic (Pose) Information

Your extrinsic parameters include:

- **Camera height:** 325 mm
- **Vertical cant (tilt):** 11° downward
- **Horizontal cant:** 0°
- **Inter - camera distance (baseline):** 240 mm

If you set up your coordinate system with the left camera at the origin, you would have:

- The left camera's rotation R_{left} includes a tilt of 11° about the horizontal axis (often the x-axis). For a rotation about the x-axis:

$$R_{\text{tilt}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos 11^\circ & -\sin 11^\circ \\ 0 & \sin 11^\circ & \cos 11^\circ \end{bmatrix}.$$

- The right camera is translated by 240 mm relative to the left camera. In a simple case (if both cameras are mounted with the same tilt) you can assume

$$t = \begin{bmatrix} 240 \\ 0 \\ 0 \end{bmatrix} \text{mm}.$$

Also, the camera height (325 mm) tells you where the cameras are relative to the ground. If you wish to compute the distance to objects on the ground (or reproject points to the ground plane) you'll later use the camera's position and orientation to intersect the viewing rays with the plane at the known height.

---

3. Stereo Rectification

Before doing any matching, you must "rectify" the images so that corresponding points lie on the same horizontal line. With the intrinsic parameters K and the extrinsic parameters (rotation and translation between the two cameras), you can use a function like OpenCV's

- `cv::stereoRectify` (C++/Python)

to compute the rectification transforms. Once you have those transforms, you can "warp" the images using functions such as `cv::initUndistortRectifyMap` and `cv::remap`.

Rectification will also handle the fact that the cameras are tilted (11° downward) so that the epipolar lines in the rectified images are horizontal.

---

4. Compute Disparity and then Depth

Once the images are rectified you use a stereo matching algorithm (for example, OpenCV's `StereoSGBM` or `StereoBM`) to compute the **disparity map**. For each corresponding point, the horizontal disparity d (in pixels) is measured.

The standard depth (or triangulation) equation is:

$$Z = \frac{f \cdot B}{d},$$

where

- Z is the depth (distance along the camera's view direction),
- f is the focal length in pixel units (here, 1440 px),
- B is the baseline (240 mm),
- d is the disparity in pixels.

Thus in your case:

$$Z\,(\text{in mm}) = \frac{1440 \times 240}{d} = \frac{345600}{d}.$$

For example, if the computed disparity at a pixel is 20 px then

$$Z = \frac{345600}{20} = 17280\,\text{mm} \quad (\approx 17.3\,\text{m}).$$

Important: This equation assumes that the images are properly rectified so that disparity is measured solely along the x-axis.

If you wish to compute not only the depth Z but the full 3D coordinates of points, you can use a triangulation function (e.g. OpenCV's `cv::triangulatePoints`) that uses the projection matrices for the two cameras (which are built from your intrinsic and extrinsic parameters).

Bringing It All Together

A typical pipeline might look like this (in pseudocode/Python with OpenCV):

```python
import cv2
import numpy as np

# --- Intrinsic parameters ---
f_px = 1440  # computed from 21.6 mm / 0.015 mm/px
K = np.array([[f_px, 0, 512],
              [0, f_px, 512],
              [0, 0, 1]], dtype=np.float64)

# --- Assume zero distortion (or provide your distortion coefficients) ---
distCoeffs = np.zeros(5)

# --- Extrinsic parameters ---
# Rotation for a downward tilt of 11° about the x-axis:
theta = np.deg2rad(11)
R_tilt = np.array([[1,          0,           0],
                   [0, np.cos(theta), -np.sin(theta)],
```

```
                [0, np.sin(theta),  np.cos(theta)]], dtype=np.float64)

# Left camera at the origin
R_left = R_tilt
t_left = np.array([[0], [0], [0]], dtype=np.float64)

# Right camera translated 240 mm along the x-axis (and same tilt)
R_right = R_tilt
t_right = np.array([[240], [0], [0]], dtype=np.float64)

# --- Image size ---
image_size = (1024, 1024)

# --- Stereo Rectification ---
R1, R2, P1, P2, Q, _, _ = cv2.stereoRectify(K, distCoeffs, K, distCoeffs,
                                            image_size, R_right @ R_left.T,
                                            t_right - t_left)

# Compute the rectification maps
map1x, map1y = cv2.initUndistortRectifyMap(K, distCoeffs, R1, P1, image_size, cv2.CV_32FC1)
map2x, map2y = cv2.initUndistortRectifyMap(K, distCoeffs, R2, P2, image_size, cv2.CV_32FC1)

# --- Load images (assumed grayscale) ---
img_left = cv2.imread('left_image.png', cv2.IMREAD_GRAYSCALE)
img_right = cv2.imread('right_image.png', cv2.IMREAD_GRAYSCALE)

# Rectify the images
rect_left = cv2.remap(img_left, map1x, map1y, cv2.INTER_LINEAR)
rect_right = cv2.remap(img_right, map2x, map2y, cv2.INTER_LINEAR)

# --- Stereo Matching ---
min_disp = 0
num_disp = 16 * 6  # must be divisible by 16
stereo = cv2.StereoSGBM_create(minDisparity=min_disp,
                               numDisparities=num_disp,
                               blockSize=7,
                               P1=8 * 3 * 7**2,
                               P2=32 * 3 * 7**2,
                               disp12MaxDiff=1,
                               uniquenessRatio=10,
                               speckleWindowSize=100,
                               speckleRange=32)
disparity = stereo.compute(rect_left, rect_right).astype(np.float32) / 16.0

# --- Depth Estimation ---
# For each pixel with disparity d:
# Z (in mm) = (1440 * 240) / d
depth_map = np.divide(345600, disparity, out=np.zeros_like(disparity), where=disparity > 0)

# Optionally, reproject to 3D using Q (which was computed during stereoRectify)
points_3D = cv2.reprojectImageTo3D(disparity, Q)
```

Final Remarks

- **Camera Tilt and Ground Plane:**
  Your cameras are mounted 325 mm above the ground and are tilted 11° downward. If you are estimating distances to objects on the ground, you can take the computed 3D points (or even work directly with the pixel coordinates and the known tilt) to compute where the viewing ray intersects the ground plane. This extra step uses the camera's extrinsic parameters (height and tilt) to “lift” the simple Z depth into a full world coordinate (e.g. computing the X, Y location on the ground).
- **Consistency of Parameters:**
  Ensure that all units are consistent. In our example, all distances are in millimeters and the focal length is converted into pixels using the pixel size. Also, verify that the sensor (or active imaging area) dimensions are consistent with your pixel count and pixel size.

By using your provided parameters in this way, you can fully calibrate your stereo rig, rectify the images, compute a disparity map, and then use triangulation (or the simple depth formula) to estimate the distance to objects in your scene.