Exercise 13: Structured light

02504 Computer vision

Morten R. Hannemose, mohan@dtu.dk, DTU Compute

May 6, 2022

Learning objectives

These exercises will introduce you to a 3D scanning pipeline using the phase shifting encoding. In this exercise you will use code for phase shift decoding, phase matching, and triangulation. Also, this time you will have some real images to play with.

3D scanning

Please download casper.zip which contains a structured light scanning of a baby T-Rex. The cameras are already calibrated and the camera calibration is in `calib.npy`. Load it by calling
`c = np.load('calib.npy', allow_pickle=True).item()`

The images are in the folder `sequence`. The first number indicates which camera it is from (0 or 1) and the second number describes the image, where

0	is an image fully illuminated by the projector (projector showing a white image)
1	is an image with the projector fully off
2-17	are the 16 images in the primary pattern shifting. This pattern has 40 periods.
18-25	are the 8 images of the secondary pattern shifting. This pattern has 41 periods.

For example, `frames1_0.png` is the image from camera 1 that is fully illuminated by the projector.

Exercise 13.1

If the cameras were not calibrated, you would need to calibrate them first.

How would you find their intrinsics?

What about the extrinsics?

Exercise 13.2

First we need to rectify the images. First run the following code to initialize the maps for rectification

```
im0 = cv2.imread("sequence/frames0_0.png")
size = (im0.shape[1], im0.shape[0])
stereo = cv2.stereoRectify(c['K0'], c['d0'], c['K1'],
                           c['d1'], size, c['R'], c['t'], flags=0)
R0, R1, P0, P1 = stereo[:4]
maps0 = cv2.initUndistortRectifyMap(c['K0'], c['d0'], R0, P0, size, cv2.CV_32FC2)
maps1 = cv2.initUndistortRectifyMap(c['K1'], c['d1'], R1, P1, size, cv2.CV_32FC2)
```

You can now rectify images by doing `cv2.remap(im, *maps, cv2.INTER_LINEAR)`, where `maps` is either `maps0` or `maps1` depending on which camera the image is coming from. This also handles the undistortion of the images. `P0` and `P1` are the projection matrices for the rectified images, which will come in handy when we need to triangulate the points in 3D.

Now for each image,

- load it,
- convert it gray-scale and floating point, and
- rectify it.

Store the resulting images in two lists (`ims0` and `ims1`), one for each camera.

Show `ims0[0]` and `ims1[0]` side-by-side and verify visually that the images have been rectified.

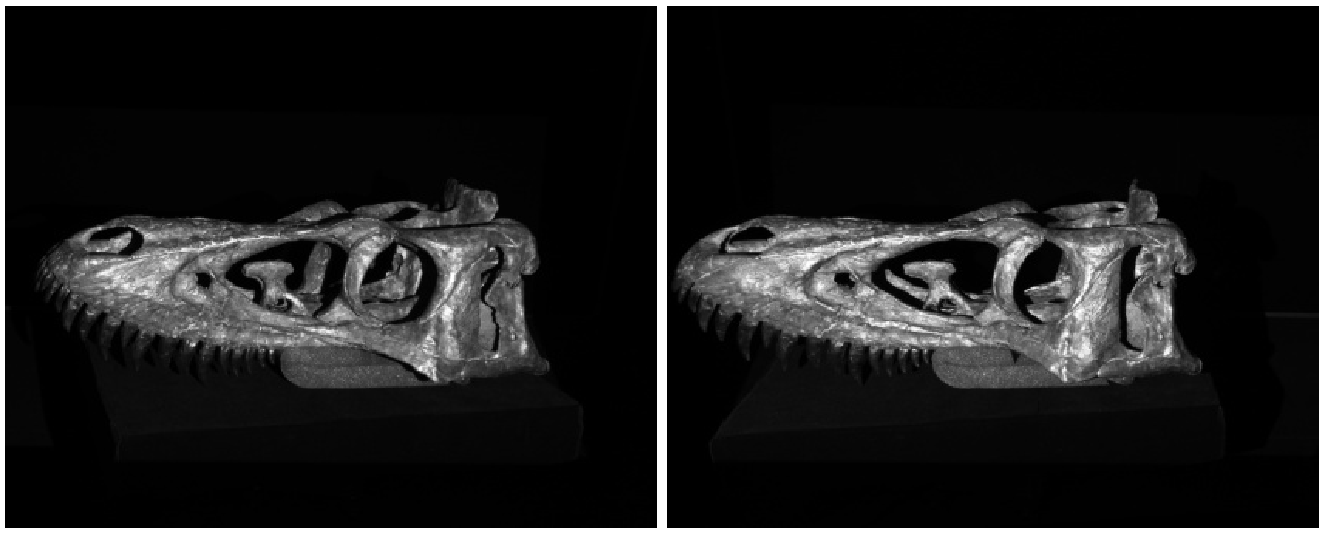
Figure 1: The rectified images with full projector light. Note how features in one image lie on a the same row in the other image, i.e. they are rectified.

Exercise 13.3

At this point, we are ready to make a function that can compute the phases for each camera `theta = unwrap(ims)`. I suggest to write the code for camera 0 and to put it into a function once it's working.

Use indexing to get a list of the primary images out, and make sure it has length 16. Put this list into the Fast Fourier Transform (`np.fft.rfft`) to find the Fourier spectrums of the primary images (`fft_primary`). We use `rfft` as the input is only real numbers. The function can operate on a list of arrays, which is ideal for our situation. Make sure to specify that the FFT should operate along the first dimension of the array (`axis=0`).

The Fourier component corresponding to the pattern is in the second component (`fft_primary[1]`). Get the phase of this using `np.angle` and call it `theta_primary`.

Repeat the same steps for the secondary phase to obtain `theta_secondary`. Compute the *phase cue* (`theta_c`) using the heterodyne principle.

Find the *order* (`o_primary`) of the primary phase.

Use the order of the primary phase to obtain the unwrapped phase (`theta`).

Wrap all of the above into a function `theta = unwrap(ims)` and use it to obtain the phase for both cameras (`theta0` and `theta1`).

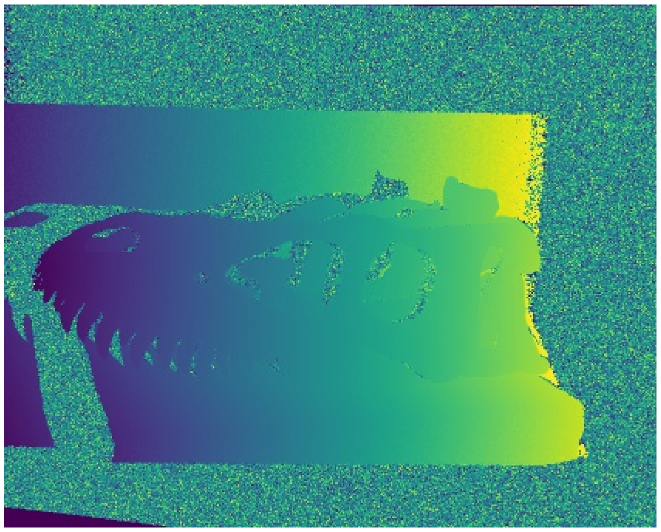

Figure 2: `theta0`

Exercise 13.4

When inspecting the phase images `theta0` and `theta1` it is clear that not all pixels contain a valid measurement of the phase. This is because some pixels do not reflect enough light from the projector to give a meaningful measurement. To fix this we introduce a binary mask that contains the areas that are sufficiently illuminated by the projector. Subtract the fully on and fully off projector image from each other (the first two elements of `ims`), to obtain a measurement of how much projector light is in each pixel.

Apply a threshold to this difference image to obtain a mask for each camera (`mask0` and `mask1`). I suggest using a threshold of 15.

Exercise 13.5

Now we need to find matches between the two cameras. As the images are rectified, we can constrain ourselves to search for a match on the corresponding row in the other image. That is we need to create two lists (`q0s` and `q1s`) that contain the pixel coordinates of matches between camera 0 and 1.

Use a double for-loop to iterate over all pixels in camera 0. For each valid pixel (`mask0[i0,j0] = True` which has the phase `theta0[i0,j0]`), we need to find the pixel in the other image that matches the best. As the images are rectified the epipolar line is a the row `i0` in camera 1. Thus, we find the matching pixel in camera 1, selecting the pixel from row `i0` which is valid `mask1[i0,j1] = True` and which has the closest phase match: `theta0[i0,j0]` $\sim$ `theta1[i0,j1]`.

We need to store the points as (x, y) i.e. `[j, i]`

To verify your matches you can compute the disparity image. This is the image such that `disparity[i0,j0] = j0-j1` for all valid pixels. Initialize it with 0 everywhere, and fill it where you have matches.

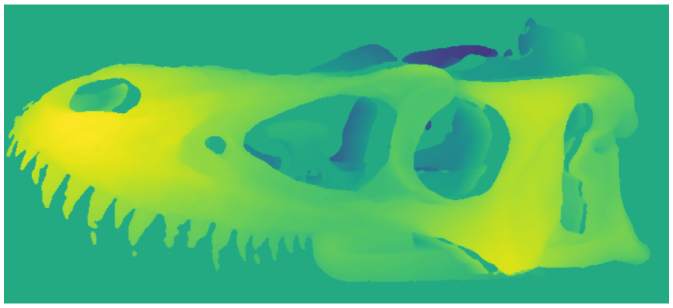

Figure 3: The disparity map visualized. Median filtering has been applied to suppress small errors throwing the colormap off, by doing `cv2.medianBlur(disparity.astype(np.float32), 5)`.

Exercise 13.6

Finally, use the matches from the previous exercise and triangulate the 3D points, using the projection matrices `P0` and `P1` from Exercise 13.2. To triangulate you can use your own function or `cv2.triangulatePoints`. For the OpenCV function you need to convert your lists to arrays, transpose them, and convert them to floating point. The output of `cv2.triangulatePoints` is in homogeneous coordinates.

To visualize the points you can use Open3D.
Install it with `pip install open3d` or `conda install -c open3d-admin open3d`

Visualize your points in 3D by doing (press [ESC] to close the window again)

```
import open3d as o3d
pcd = o3d.geometry.PointCloud()
pcd.points = o3d.utility.Vector3dVector(Q.T)
o3d.visualization.draw_geometries([pcd])
```

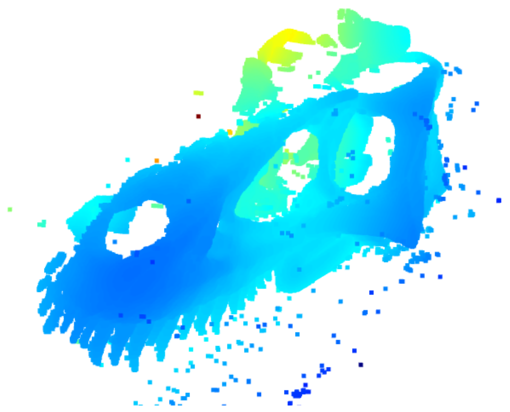
Figure 4: A visualization of the resulting 3D point cloud.

Optional exercises

Exercise 13.7

To make the point cloud more visually pleasing, we want to add color information to it.

As we know which points correspond to which matches, we can sample the color of the points directly in the image. Add this color information to the point cloud visualization.

Exercise 13.8

In Exercise 13.5, you found the camera 1 pixels which were closest to the camera 0 pixels. This is a discrete result i.e. only integer pixels will return. However, we can perform linear interpolation of phases between pixels in camera 1. In this case, sub-pixel precision is obtained by finding the sub-pixel position between two pixels, where the phases match exactly.

Try to implement sub-pixel precision in the your matcher function, and see if you get a less noisy result.