Lab5

1 Basic part

1.1 Introduction

This lab implements a projector–camera-based stereo vision pipeline using Gray-code structured light and a calibrated stereo camera rig. Instead of matching image textures directly, we project binary Gray-code patterns, decode for each camera the per-pixel projector coordinates (xp, yp), and then establish left–right correspondences by matching equal (xp, yp). Finally, we triangulate the matched pixels using the stereo calibration (R, T) to reconstruct a dense, accurate 3D point cloud.

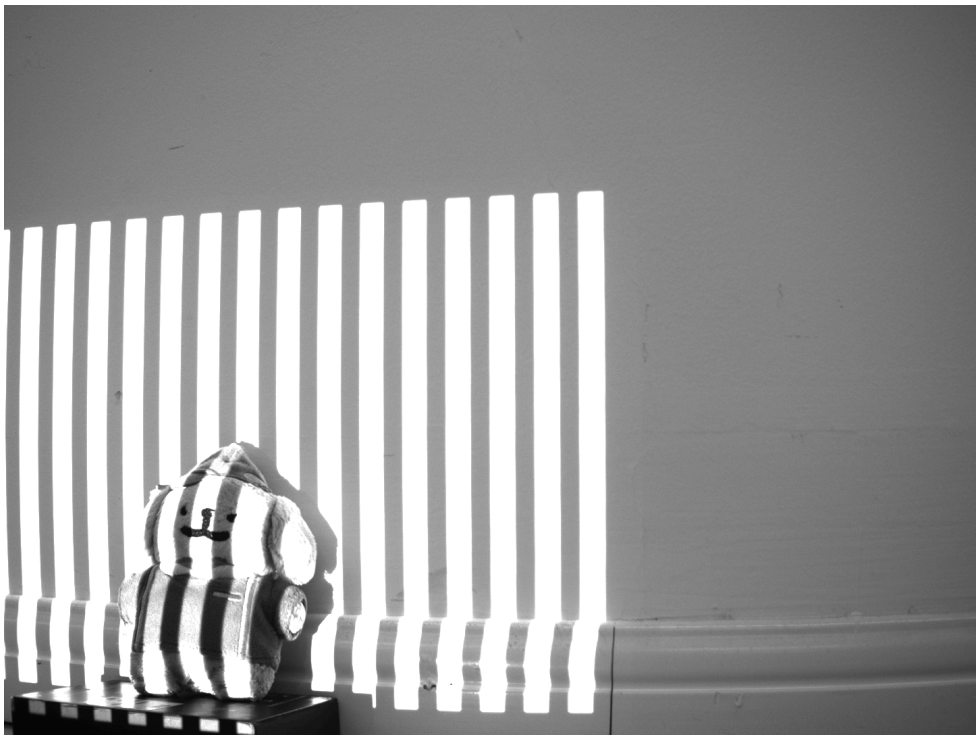
1.2 Experiment Setup

Equipment: two Firefly FFY-U3-16S2M camera

Projector: 1280×720 DLP projector aligned to illuminate the scene.

1.2.1 method of grabbing images

We use two monochrome Firefly cameras mounted rigidly on a stereo bar, aimed at the scene illuminated by a DLP projector. Image acquisition was performed with our `grap.py` helper, which previews both streams and saves synchronized pairs on keypress. Example captured frames are shown below:



For pattern projection we generated Gray-code sequences (`graycode.py`) and displayed them full-screen to the projector (`project.py`) in either manual (space to advance) or timed auto mode. Each capture session records one

reference white, one reference black, followed by MSB→LSB bit images for the vertical (x) and horizontal (y) codes per camera.

Key code excerpt (capture loop, `grap.py`):

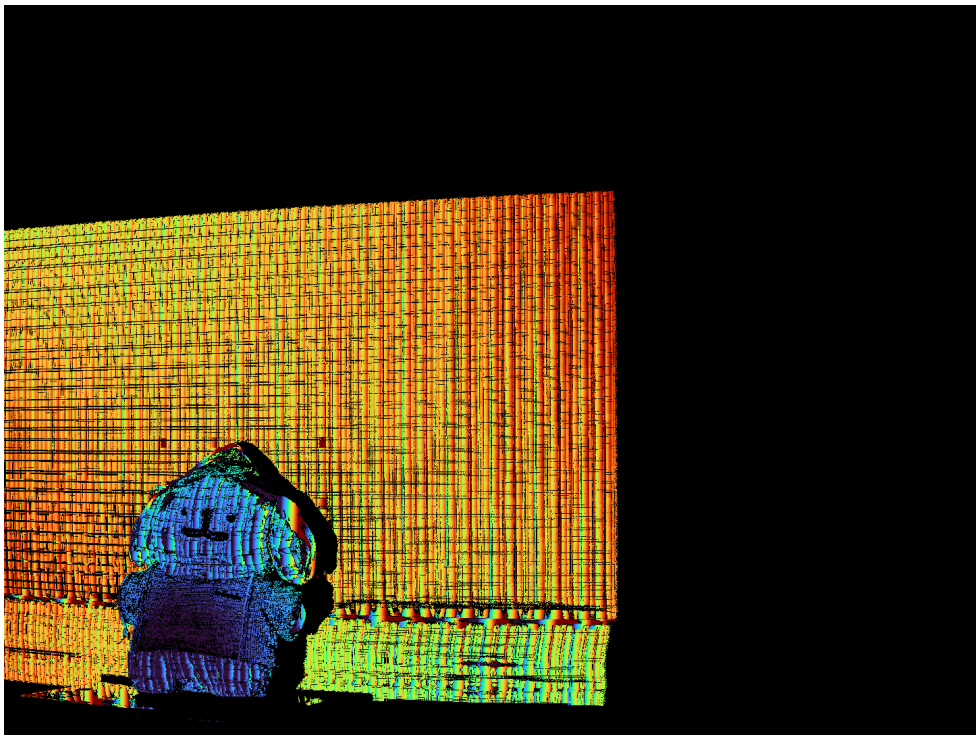
```
# Acquire frames, preview, save on SPACE/'y'
left_img = left_cam.GetNextImage(2000)
right_img = right_cam.GetNextImage(2000)
left_arr = left_img.GetNDArray(); right_arr = right_img.GetNDArray()
cv2.imshow('Left', left_arr); cv2.imshow('Right', right_arr)
key = cv2.waitKey(1) & 0xFF
if key == 32 or key == ord('y'):
        cv2.imwrite(f'{out_root}/{idx}_left.png', left_arr)
        cv2.imwrite(f'{out_root}/{idx}_right.png', right_arr)
        idx += 1
```

1.3 Result and Data Processing

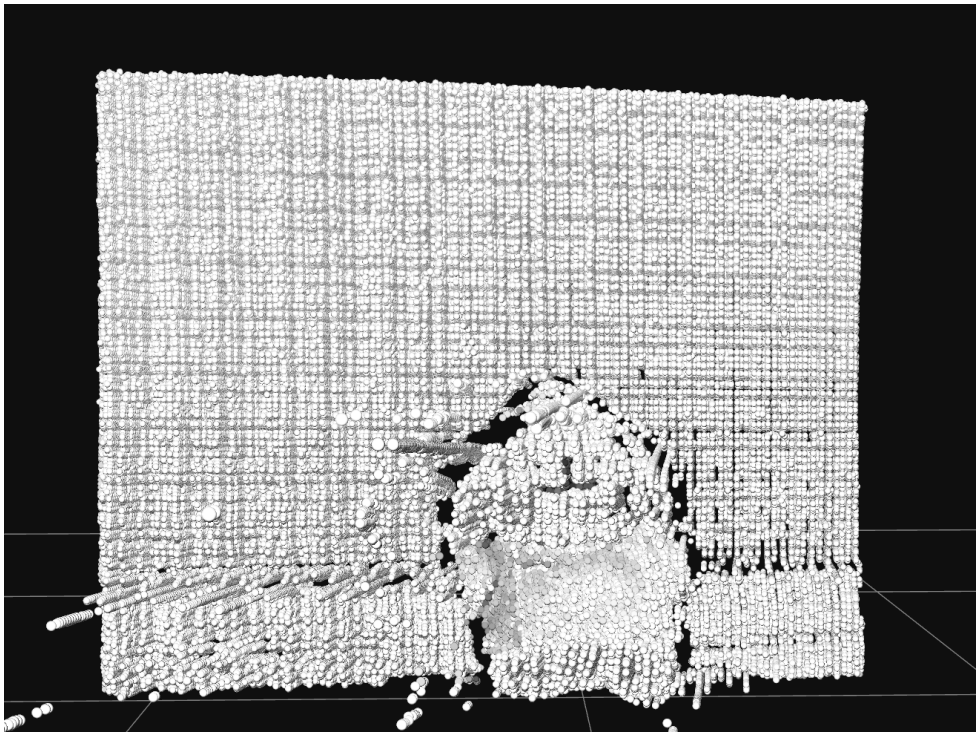
1.3.2 The quality of Depth Map and Point Cloud

We reconstruct the 3D structure by first decoding projector coordinates and then triangulating matched pixels across the stereo pair using the calibrated extrinsics (R, T). The depth map is rendered with a Turbo colormap for visualization, and the corresponding 3D point cloud is exported as a PLY for inspection.

Depth map (pseudo-color):



Point cloud (Because out camera is **gray-scale**, the point cloud is also gray-scale):



Qualitatively, the depth map exhibits consistent gradients across planar regions and preserves depth discontinuities along object boundaries. The point cloud reveals clean planar structures and reasonable surface continuity. Minor holes correspond to shadowed areas or pixels filtered by the decoding reliability mask.

Key code excerpt (matching + triangulation + depth, `reconstruct.py`):

```
def build_right_map(rx_map, ry_map, rmask):
        mp = defaultdict(list)
        h, w = rx_map.shape
        for y in range(h):
                xs = np.where(rmask[y] > 0)[0]
                for x in xs:
                        mp[((rx_map[y,x]), (ry_map[y,x]))].append((x, y))
        return mp

# For each valid left pixel, find right candidate(s) by (xp, yp)
for y in range(h):
        xs = np.where(lmask[y] > 0)[0]
        for x in xs:
                key = (int(lx[y,x]), int(ly[y,x]))
                cand = right_map.get(key, [])  # or tolerant search
                if not cand: continue
                rx_med = int(np.median([c[0] for c in cand]))
                ry_med = int(np.median([c[1] for c in cand]))
        pts_left.append((x, y)); pts_right.append((rx_med, ry_med))

# Undistort to normalized coords and triangulate
P1 = np.hstack([np.eye(3), np.zeros((3,1))])
P2 = np.hstack([R, T])
X_h = cv2.triangulatePoints(P1, P2, pts1_ud.T, pts2_ud.T)
X = (X_h[:3, :] / X_h[3, :]).T  # Nx3

# Build depth (meters), auto-handle sign/unit then visualize
depth = np.full((h, w), np.nan, np.float32)
zz = X[:,2]
if np.nanmedian(zz) < 0: zz = -zz

z_m = zz * scale_to_m
```

```
valid = np.isfinite(z_m) & (z_m > 0)
pl = np.array(pts_left, np.int32)
depth[pl[valid,1], pl[valid,0]] = z_m[valid]
```

1.3.3 Decoding Gray-code to projector coordinates

Given a set of captured images per camera consisting of reference white/black and Gray-code bit planes (vertical for column index, horizontal for row index), we perform the following steps:

- Compute a shadow/reliability mask by comparing white vs black frames to suppress pixels with insufficient illumination contrast.
- For each axis, compute the per-pixel mid level and decode MSB→LSB bit images into Gray code, then convert to binary indices to recover projector coordinates (xp, yp).
- Enforce validity by checking index ranges and combining with the reliability mask to produce `*_mask.png`.
- Save float maps `left_x.tiff/left_y.tiff` and `right_x.tiff/right_y.tiff` along with visualization PNGs.

We then create a right-image lookup: for each valid right pixel, map its (xp, yp) to the pixel location. For every valid left pixel, we find the corresponding right pixel(s) with matching (xp, yp) (exact or within a small tolerance) and take the median if multiple candidates exist. These correspondences are triangulated with the stereo calibration to recover 3D points.

Key code excerpt (decoding core, `decode.py`):

```
def gray_to_binary_arr(gray: np.ndarray, nbits: int) -> np.ndarray:
        b = np.zeros_like(gray); g = gray.copy()
        for _ in range(nbits):
                b ^= g; g >>= 1
        return b

def decode_axis(patterns, white, black, axis_bits, white_thresh):
        mid = ((white.astype(np.int32) + black.astype(np.int32)) // 2)
        gray_vals = np.zeros_like(mid)
        reliable = np.ones_like(mid, dtype=bool)
        # MSB → LSB
        for bit_i in range(axis_bits):
                img = patterns[bit_i].astype(np.int32)
```

```
                reliable &= (np.abs(img - mid) > int(white_thresh))
                shift = (axis_bits - 1 - bit_i)
                gray_vals |= ((img > mid).astype(np.int32) << shift)
        idx = gray_to_binary_arr(gray_vals, axis_bits)
        return idx, reliable
```

2 Discussion

Structured light with Gray-code provides discrete, robust correspondences that are largely invariant to local surface texture, avoiding many failure modes of passive stereo in low-texture regions. The main challenges observed are:

- Shadows and low SNR: regions not sufficiently illuminated or affected by inter-reflections fail the white–black contrast test and become invalid (holes in depth).
- Decoding ambiguities near bit transitions: pixels whose intensity is close to the mid-level for a given bit are marked unreliable to prevent code flips.
- Projector–camera geometry coverage: oblique incidence and occlusions reduce valid area; careful projector alignment expands overlap with both cameras.

To improve coverage and stability we used a conservative reliability threshold and allowed a small tolerance when matching (xp, yp) across views; this trades a few ambiguities for better completeness, mitigated by median aggregation of candidates. We also export float depth (meters) and a uint16-mm depth for tooling compatibility; invalids are kept as NaN (float) or 0 (PNG) to make failure regions explicit.

3 Conclusion

We built a practical projector–camera-based stereo pipeline that leverages Gray-code decoding for correspondence and a standard stereo calibration for accurate triangulation. The method avoids geometry mismatch issues and produces stable point clouds with clear planar structures. With minor parameter tuning and robust masking, the approach is reliable across varied scenes.

References

we chat with AI for some important problems.