Yash Gopani – CS532 Homework Assignment 1

Part 1: Image Warping

(a) DLT using Point Correspondences

This section applies the Direct Linear Transform (DLT) algorithm to compute a homography using four manually selected point correspondences between the original and destination images. The computed homography matrix is used for inverse warping to generate a top-down rectified view.

Listing 1: Python Implementation of DLT using Point Correspondences

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

class HomographyDLTPoints:
    def __init__(self, input_path):
        self.input_path = input_path
        self.enhanced_img = None

    def preprocess(self):
        img = np.array(Image.open(self.input_path).convert("RGB")
            ↪ )
        lab = cv2.cvtColor(img, cv2.COLOR_RGB2LAB)
        l, a, b = cv2.split(lab)
        clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,
            ↪ 8))
        l = clahe.apply(l)
        self.enhanced_img = cv2.cvtColor(cv2.merge((l, a, b)),
            ↪ cv2.COLOR_LAB2RGB)

    def compute_homography(self, src, dst):
        A = []
        for (x, y), (u, v) in zip(src, dst):
            A.append([-x, -y, -1, 0, 0, 0, u * x, u * y, u])
            A.append([0, 0, 0, -x, -y, -1, v * x, v * y, v])

        _, _, Vt = np.linalg.svd(np.array(A))
        H = Vt[-1].reshape(3, 3)
        return H / H[2, 2]

    def warp_image(self, H, output_size):
        warped = cv2.warpPerspective(self.enhanced_img, H,
            ↪ output_size)
        return warped
```

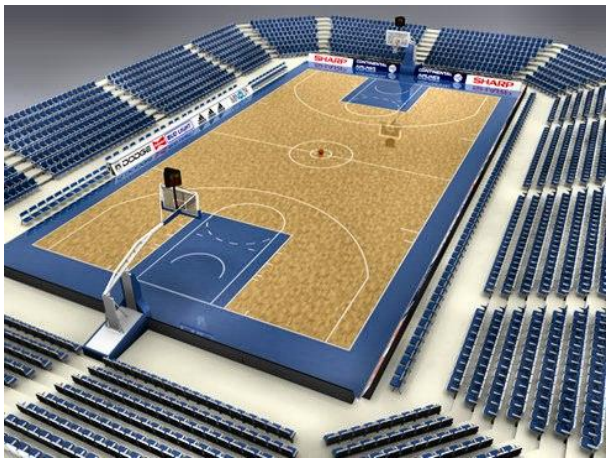

Figure 1: Original Image of Basketball Court

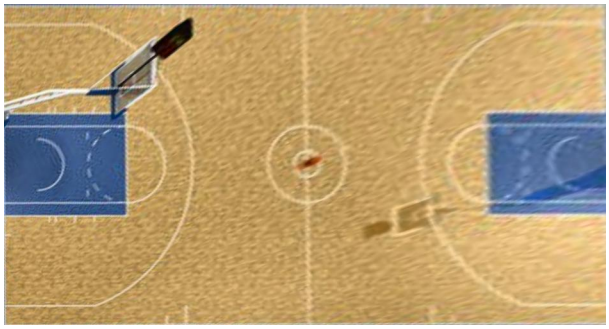

Figure 2: Warped Image using DLT Point Correspondences

(b) Bilinear Interpolation

To improve visual quality and avoid aliasing during inverse warping, bilinear interpolation was implemented to estimate pixel values between integer coordinates.

Listing 2: Python Implementation of Bilinear Interpolation for Inverse Warping

```
import cv2
import numpy as np
from PIL import Image
import os

class BilinearWarp:
    def __init__(self, input_path):
        self.input_path = input_path
        self.enhanced_img = None
        os.makedirs("images/output", exist_ok=True)

    def preprocess(self):
```

```
        img = np.array(Image.open(self.input_path).convert("RGB")
            ↪ )
        lab = cv2.cvtColor(img, cv2.COLOR_RGB2LAB)
        l, a, b = cv2.split(lab)
        l = cv2.createCLAHE(2.0, (8, 8)).apply(l)
        self.enhanced_img = cv2.cvtColor(cv2.merge((l, a, b)),
            ↪ cv2.COLOR_LAB2RGB)

    def bilinear(self, img, x, y):
        x0, y0 = int(x), int(y)
        x1, y1 = min(x0 + 1, img.shape[1] - 1), min(y0 + 1, img.
            ↪ shape[0] - 1)
        dx, dy = x - x0, y - y0
        tl, tr, bl, br = img[y0, x0], img[y0, x1], img[y1, x0],
            ↪ img[y1, x1]
        return ((1 - dy) * ((1 - dx) * tl + dx * tr) + dy * ((1 -
            ↪  dx) * bl + dx * br)).astype(np.uint8)

    def warp(self, H, size=(940, 500)):
        Hinv = np.linalg.inv(H)
        out = np.zeros((size[1], size[0], 3), dtype=np.uint8)
        for y in range(size[1]):
            for x in range(size[0]):
                X = Hinv @ np.array([x, y, 1])
                if X[2] != 0:
                    sx, sy = X[0] / X[2], X[1] / X[2]
                    if 0 <= sx < self.enhanced_img.shape[1] and 0
                        ↪  <= sy < self.enhanced_img.shape[0]:
                        out[y, x] = self.bilinear(self.
                            ↪ enhanced_img, sx, sy)
        return out
```

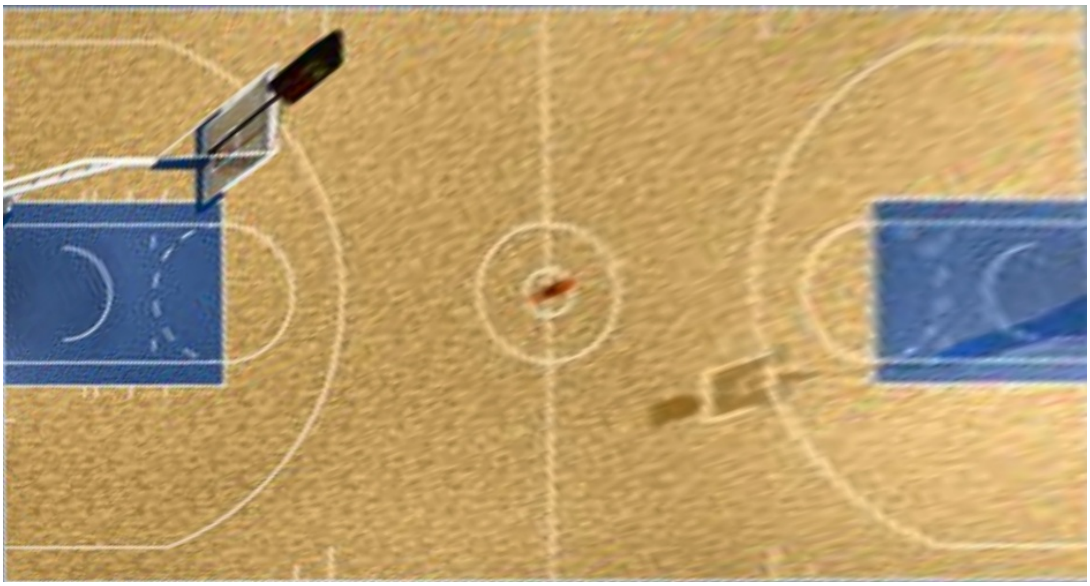

Figure 3: Result of Bilinear Interpolation on Warped Image

- The bilinear interpolation smooths out jagged edges seen in nearest-neighbor warping.
- CLAHE enhancement improves local contrast before warping.
- This method produces a continuous tone-mapped image with minimal pixel artifacts.

(c) DLT using Line Feature Locations

Here, the homography matrix was computed using four boundary line correspondences (top, bottom, left, right) instead of corner points. The system of equations was solved using Singular Value Decomposition (SVD) for numerical stability.

Listing 3: DLT using Line Features (Part c)

```
class LineHomographyDLT:
    def __init__(self, input_path):
        self.input_path = input_path
        self.img = preprocess_image(input_path, "images/output/
            ↪ enhanced_partC.jpg")

    def line_eq(self, p1, p2):
        x1, y1 = p1; x2, y2 = p2
        a, b, c = y1 - y2, x2 - x1, x1 * y2 - x2 * y1
        norm = math.hypot(a, b)
        return np.array([a, b, c]) / norm if norm != 0 else np.
            ↪ array([a, b, c])

    def compute_line_homography(self, src_lines, dst_lines):
        A = []
        for src_line_pts, dst_line_pts in zip(src_lines,
            ↪ dst_lines):
            l_src = self.line_eq(src_line_pts[0], src_line_pts
                ↪ [1])
            l_dst = self.line_eq(dst_line_pts[0], dst_line_pts
                ↪ [1])
            row1 = [0,0,0, -l_src[2]*l_dst[0], -l_src[2]*l_dst
                ↪ [1], -l_src[2]*l_dst[2],
                    l_src[1]*l_dst[0], l_src[1]*l_dst[1], l_src
                        ↪ [1]*l_dst[2]]
            row2 = [l_src[2]*l_dst[0], l_src[2]*l_dst[1], l_src
                ↪ [2]*l_dst[2], 0,0,0,
                    -l_src[0]*l_dst[0], -l_src[0]*l_dst[1], -
                        ↪ l_src[0]*l_dst[2]]
            A.append(row1); A.append(row2)
        _, _, Vt = np.linalg.svd(np.array(A))
        H_T = Vt[-1].reshape(3,3)
        H = H_T.T
        return H / H[2,2]
```

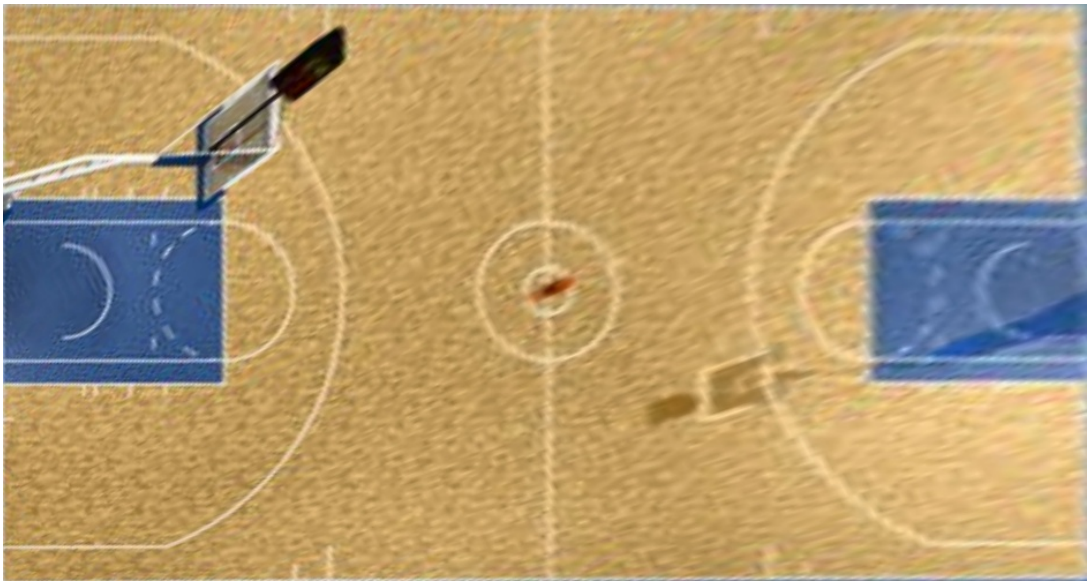

Figure 4: Rectified Image using DLT from Line Feature Correspondences

- Line-based DLT provided a robust geometric alignment when point correspondences were noisy.
- The computed homography was validated by visualizing rectified parallel boundaries.

Part 2: Dolly Zoom (Object-Centered Motion)

Click here to view Dolly Zoom Video on GitHub

Listing 4: Python Implementation of Dolly Zoom (Fish Object-Centered Motion)

```
# Camera path and rendering using object-centered motion
def generate_camera_poses(object_center, viewing_distance,
    ↪ num_frames):
    angles = np.linspace(-np.pi / 2, np.pi / 2, num_frames)
    poses = []
    for angle in angles:
        cam_x = viewing_distance * np.cos(angle)
        cam_z = viewing_distance * np.sin(angle)
        cam_pos = np.array([[cam_x], [0], [cam_z]])
        forward = (object_center - cam_pos).flatten()
        forward /= np.linalg.norm(forward)
        up = np.array([0, 1, 0])
        right = np.cross(up, forward)
        right /= np.linalg.norm(right)
        up = np.cross(forward, right)
        R = np.vstack([right, up, forward])
        t = -R @ cam_pos
        poses.append((R, t))
    return poses
```

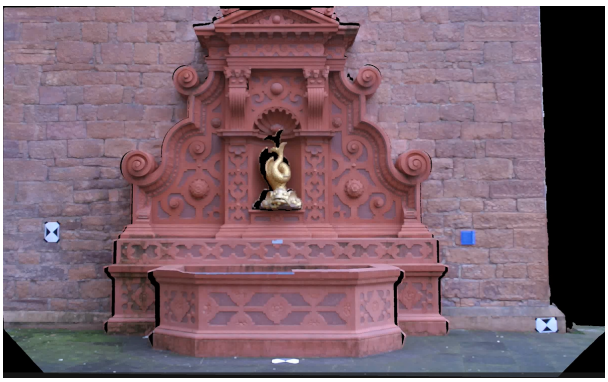

Figure 5: Initial Frame – Camera at Start of Half-Circle Path

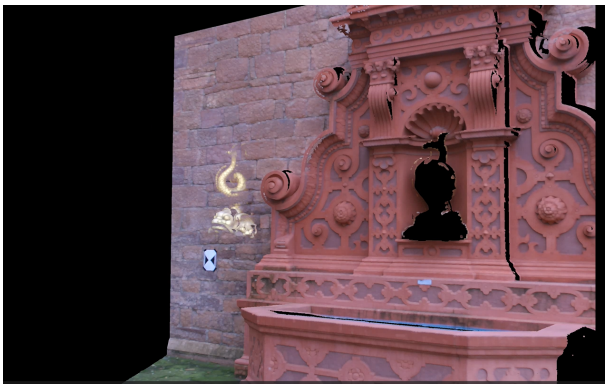

Figure 6: Mid Sequence – Dolly Zoom Effect Maintaining Constant Object Size

Figure 7: Final Frame – Camera at End of Trajectory

Observations and Notes

- The dolly zoom was achieved by dynamically adjusting the camera's distance and orientation to the object while maintaining constant projection size.
- The half-circle trajectory ensures a smooth orbit around the object's centroid.
- Point cloud rendering used both background and foreground layers for photorealistic results.

Conclusion

This report demonstrates a comprehensive understanding of homography estimation, image warping, and perspective transformations. The DLT methods successfully rectified planar scenes, while the Dolly Zoom implementation simulated realistic motion and depth variation using the pinhole camera model.