CS532 Assignment: 1

Problem 1

Source code:

```
% main.m
function main()
   bcg_im = imread('basketball-court.ppm');  % Load the original image
   bcp_im = uint8(zeros(500, 940, 3));  % Create an empty output image

   bcp_im = HW_P1(bcg_im, bcp_im);  % Call the function with both input images

   imshow(bcp_im);
   title('Warped Output Image');
   imwrite(bcp_im, 'warped_output_image.jpg');
end

% HW_P1.m
function bcp_im = HW_P1(bcg_im, bcp_im)
   bcg_im_ar = double(bcg_im);

   [row, col, ~] = size(bcg_im_ar);

   % Define the source and destination points
   source_ps = [0, 190; 245, 45; 420, 70; 290, 300];
   dstn_ps = [0, 0; 939, 0; 939, 499; 0, 499];

   % Define the source and destination lines
   source_lines = [0, 0, 500, 0; 500, 0, 500, 500; 500, 500, 0, 500; 0, 500,
0, 0];
   destination_lines = [0, 0, 939, 0; 939, 0, 939, 499; 939, 499, 0, 499; 0,
499, 0, 0];
```

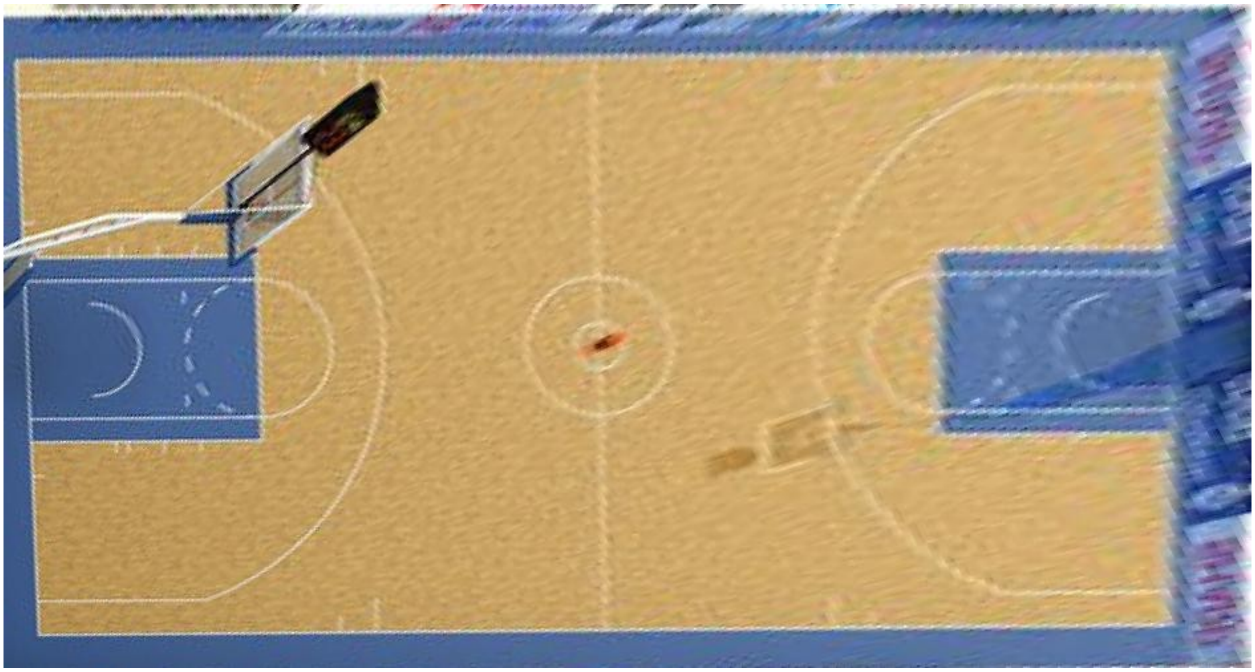
```matlab
% Compute homography using point correspondences
H_points = dlt(source_ps, dstn_ps);

% Compute homography using line correspondences
H_lines = dlt_line(source_lines, destination_lines);

% Choose one of the homography matrices (you can experiment with both)
H = H_points; % or H = H_lines;

% Calculate the inverse of the homography matrix
H_inv = inv(H);

% Iterate over every pixel in the output image
for i = 1:size(bcp_im, 1)
    for j = 1:size(bcp_im, 2)
        bcp_p = [j; i; 1];
        bcr_p = H_inv * bcp_p;
        bcr_p = bcr_p / bcr_p(3);
        bcx = round(bcr_p(1));
        bcy = round(bcr_p(2));

        % Perform bilinear interpolation
        if bcx >= 1 && bcy >= 1 && bcx <= col && bcy <= row
            dx = bcr_p(1) - bcx;
            dy = bcr_p(2) - bcy;
            x1 = min(max(bcx, 1), col);
            y1 = min(max(bcy, 1), row);
            x2 = min(max(bcx + 1, 1), col);
            y2 = min(max(bcy, 1), row);
```

```
                x3 = min(max(bcx, 1), col);
                y3 = min(max(bcy + 1, 1), row);
                x4 = min(max(bcx + 1, 1), col);
                y4 = min(max(bcy + 1, 1), row);

                red = (1 - dx) * (1 - dy) * bcg_im_ar(y1, x1, 1) + ...
                      dx * (1 - dy) * bcg_im_ar(y2, x2, 1) + ...
                      (1 - dx) * dy * bcg_im_ar(y3, x3, 1) + ...
                      dx * dy * bcg_im_ar(y4, x4, 1);

                green = (1 - dx) * (1 - dy) * bcg_im_ar(y1, x1, 2) + ...
                        dx * (1 - dy) * bcg_im_ar(y2, x2, 2) + ...
                        (1 - dx) * dy * bcg_im_ar(y3, x3, 2) + ...
                        dx * dy * bcg_im_ar(y4, x4, 2);

                blue = (1 - dx) * (1 - dy) * bcg_im_ar(y1, x1, 3) + ...
                       dx * (1 - dy) * bcg_im_ar(y2, x2, 3) + ...
                       (1 - dx) * dy * bcg_im_ar(y3, x3, 3) + ...
                       dx * dy * bcg_im_ar(y4, x4, 3);

                bcp_im(i, j, :) = uint8([red, green, blue]);
            end
        end
    end
end

% DLT Algorithm for Homography Estimation Using Point Correspondences
function H = dlt(source_points, destination_points)
    num_points = size(source_points, 1);
    if num_points ~= size(destination_points, 1)
```

```
        error('Number of source and destination points must be equal');
    end

    % Construct the coefficient matrix A for DLT
    A = [];
    for i = 1:num_points
        x = source_points(i, 1);
        y = source_points(i, 2);
        xd = destination_points(i, 1);
        yd = destination_points(i, 2);

        A = [A; -x, -y, -1, 0, 0, 0, x*xd, y*xd, xd;
                 0, 0, 0, -x, -y, -1, x*yd, y*yd, yd];
    end

    % Perform Singular Value Decomposition (SVD) of A
    [~, ~, V] = svd(A);

    % Extract the solution (last column of V)
    H = reshape(V(:, end), 3, 3)';
end

% DLT Algorithm for Homography Estimation Using Line Feature Locations
function H = dlt_line(source_lines, destination_lines)
    num_lines = size(source_lines, 1);
    if num_lines ~= size(destination_lines, 1)
        error('Number of source and destination lines must be equal');
    end

    % Construct the coefficient matrix A for DLT
```

```
    A = [];
    for i = 1:num_lines
        % Extract source line endpoints
        x1s = source_lines(i, 1);
        y1s = source_lines(i, 2);
        x2s = source_lines(i, 3);
        y2s = source_lines(i, 4);

        % Extract destination line endpoints
        x1d = destination_lines(i, 1);
        y1d = destination_lines(i, 2);
        x2d = destination_lines(i, 3);
        y2d = destination_lines(i, 4);

        % Construct equations for line homography
        Ai = [0, 0, 0, -x1s, -y1s, -1, y1d*x1s, y1d*y1s, y1d;
              x1s, y1s, 1, 0, 0, 0, -x1d*x1s, -x1d*y1s, -x1d;
              0, 0, 0, -x2s, -y2s, -1, y2d*x2s, y2d*y2s, y2d;
              x2s, y2s, 1, 0, 0, 0, -x2d*x2s, -x2d*y2s, -x2d];

        A = [A; Ai];
    end

    % Perform Singular Value Decomposition (SVD) of A
    [~, ~, V] = svd(A);

    % Extract the solution (last column of V)
    H = reshape(V(:, end), 3, 3)';
end
```

Output image



main.m: This is the main script where the program starts execution. It loads the original image, calls the **HW_P1** function to perform image warping, displays the warped image, and saves it as "warped_output_image.jpg".

HW_P1.m: This function performs the image warping. Here's what it does.It loads the original image and initializes an empty output image.It defines source and destination points, as well as source and destination lines.It computes the homography matrices using both point correspondences (**source_ps** and **dstn_ps**) and line correspondences (**source_lines** and **destination_lines**) using the DLT algorithm.It chooses one of the computed homography matrices (**H_points** or **H_lines**) for the transformation. You can experiment with both methods.It calculates the inverse of the chosen homography matrix (**H_inv**).It iterates over every pixel in the output image and performs the inverse transformation using bilinear interpolation to map pixels from the original image to the output image.

DLT Algorithm for Homography Estimation Using Point Correspondences (dlt): This function computes the homography matrix using point correspondences. It constructs the coefficient matrix **A** for the DLT algorithm and performs Singular Value Decomposition (SVD) to obtain the homography matrix **H**.

DLT Algorithm for Homography Estimation Using Line Feature Locations (dlt_line): This function computes the homography matrix using line feature locations. Similar to the point-based DLT, it constructs the coefficient matrix **A** for the DLT algorithm using equations derived from line correspondences and performs SVD to obtain the homography matrix **H**.This code demonstrates how to perform image warping using homography matrices computed from both point and line correspondences, providing flexibility in feature selection for the transformation.

Problem 2

Source code

```
% Load necessary data

load data.mat  % Contains BackgroundPointCloudRGB, ForegroundPointCloudRGB, K,
crop_region, filter_size


% Extract the foreground object (fish statue) point cloud

foregroundPointCloud = ForegroundPointCloudRGB;


% Calculate the center of the foreground object

foregroundCenter = mean(foregroundPointCloud(1:3, :), 2);


% Define parameters for the half-circle path

radius = 2;  % Distance from the center of the foreground object

numFrames = 50;  % Number of frames in the capture sequence


% Generate camera positions along the half-circle path

theta = linspace(pi, 2*pi, numFrames);  % Angle range for half circle

cameraPositions = [foregroundCenter(1) + radius * cos(theta);

                  foregroundCenter(2) + radius * sin(theta);

                  foregroundCenter(3) * ones(1, numFrames)];

% Initialize cell array to store rendered images

renderedImages = cell(1, numFrames);


% Iterate over each camera position and render the image

for frame = 1:numFrames
```

```
   % Calculate extrinsic parameters for the current frame
   % Translation: Move camera to the current position
   t = cameraPositions(:, frame);
   % Rotation: Keep the camera looking at the center of the foreground object
   % You can adjust the rotation method based on your specific requirements
   R = eye(3);  % Identity rotation for simplicity

   % Construct the projection matrix
   M = K * [R t];

   % Render the image for the current frame
   renderedImages{frame} = PointCloud2Image(M, {BackgroundPointCloudRGB,
ForegroundPointCloudRGB}, crop_region, filter_size);
end

% Save or display the rendered images as needed
% For example, save each image to a file
for frame = 1:numFrames
   fname = sprintf('frame_%03d.jpg', frame);
   imwrite(renderedImages{frame}, fname);
end



% Initialize VideoWriter object
videoFilename = 'Problem2.avi';
writerObj = VideoWriter(videoFilename, 'MPEG-4');
writerObj.FrameRate = 10;  % Adjust the frame rate as needed
open(writerObj);

% Iterate over each rendered image and write it to the video
```

```
for frame = 1:numFrames
   % Write the frame to the video
   writeVideo(writerObj, renderedImages{frame});
end

% Close the VideoWriter object
close(writerObj);

disp(['Video saved as ' videoFilename]);
```