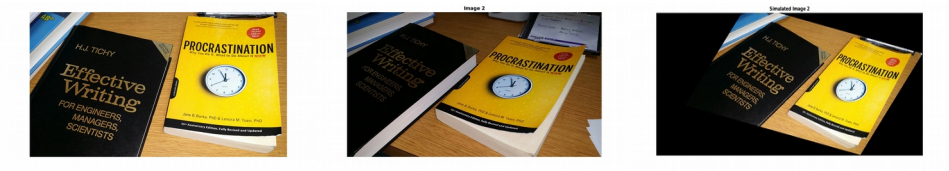
Matt Boler
ELEC 7450 – Project 5

Affine transforms are characterized by a combination of a matrix multiplication and an additive offset of the original coordinates of the form y = Ax + b. When implementing an affine transformation function, an important nuance is the use of backwards mapping versus forwards mapping. Forwards mapping, in which the value of each input pixel is transformed forward into the output image, can both be inefficient and produce a suboptimal image as input pixels may not explicitly map to every output pixel. Backwards mapping, on the other hand, finds a suitable value for every output pixel, ensuring a reasonable output image. A note on the implementation of backwards mapping: When inverting the pixel coordinates of the output image, some pixels may map outside the bounds of the input image. In this implementation this is handled by zero-padding the input image and adjusting internal coordinates accordingly. Also, in the interest of preserving all information the implementation includes the translation as zero-padding as appropriate. If this behavior is undesirable it is trivial to un-pad as the padding is a function of the input vector b. Below are shown the input image, the image after a forward-mapped affine transform, and the image after a backward-mapped affine transform. Clearly the backwards map produces a superior image. This particular transform (A = [0.5, 0.5; 0.5, -1.5], b = [0; 0]) mirrors the image about the horizontal centerline, rotates the image clockwise, and stretches it vertically/compresses it horizontally.


An unknown transformation between two images can be estimated as well. By finding the pixel coordinates of corresponding points in both images and performing a least-squares fit. This fit is performed in two steps; one for the transforming of x coordinates and one for the transforming of y coordinates. In both cases, the problem is formulated as [output] = [x_in, y_in, 1]*[a; b; c], where output is the stacked vector of output coordinates and x_in and y_in are the stacked vectors of input pixels. A demonstration of this process is shown below, where the transform between the first and second images is estimated and the third image shows the result of applying the estimated transform. The result is not a perfect match; this is because the movement of a camera is actually described by projective transforms rather than affine transforms, which require 3D information of the scene. The affine transform implicitly ignores depth, and as such doesn't account for the change in perspective.




The code for implementing a backward-mapped affine transform is included below:

```
function [outImage] = affineTransformBackward(t_matrix, t_vector, t_image)
```

```
%AFFINETRANSFORMBACKWARD Summary of this function goes here
%   Detailed explanation goes here

[h, w, d] = size(t_image);
du_in = floor(w/2);
dv_in = floor(h/2);

T = [t_matrix, t_vector; 0 0 1];
T_inv = inv(T);
A_back = T_inv(1:2, 1:2);
b_back = T_inv(1:2, 3);

% Part 1: Generate empy output image
% Transform corners of input image to find bounds of output image

du = floor(w/2);
dv = floor(h/2);
inputCorners = [0 0 w w; 0 h 0 h] - [du; dv];
outputCorners = t_matrix * inputCorners + t_vector;

minOutU = floor(min(outputCorners(1,:)));
minOutV = floor(min(outputCorners(2,:)));
maxOutU = ceil(max(outputCorners(1,:)));
maxOutV = ceil(max(outputCorners(2,:)));

% This is the size of the area covered by the transformed input image
h_out = maxOutV - minOutV + 1;
w_out = maxOutU - minOutU + 1;
% Ensure all of input image fits in output image (results in zero padding when translation occurs)
% If zero-padding isn't desired, un-pad by these same values after mapping is performed
outImage = zeros(h_out + ceil(t_vector(2)), w_out + ceil(t_vector(1)), d);

% Part 2: Pad input image with reflected copies of original
% Backwards warp corners of output to find where/if they exceed input image

zero_image = zeros(size(t_image));

top_bot = zero_image;
left_right = zero_image;
corners = zero_image;
padded_image = [corners, top_bot, corners; left_right, t_image, left_right; corners, top_bot,
corners];

% Part 3: Backwards map!

outCoords = getCoordinates(outImage);
du_out = floor(w_out/2);
dv_out = floor(h_out/2);
outCoords_centered = outCoords - [du_out; dv_out];

for i = 1:size(outCoords_centered, 2)
   u = outCoords_centered(1,i);
   v = outCoords_centered(2,i);

   p = [u; v];
   p_inv = A_back * p + b_back;
   % Coords in uncentered input image
   u_in = round(p_inv(1) + du_in + w);
```

```
   v_in = round(p_inv(2) + dv_in + h);

   outImage(v+dv_out, u+du_out, :) = padded_image(v_in, u_in, :);
end

outImage = uint8(outImage);

end

function coords = getCoordinates(t_input)
% This function generates a [u; v] coordinate vector for each pixel in an
% image
    [h, w, ~] = size(t_input);
    [U, V] = meshgrid(1:w, 1:h);
    U_row = reshape(U, 1, numel(U));
    V_row = reshape(V, 1, numel(V));
    coords = [U_row; V_row];
end
```