CSci 5563 Assignment 2

Luis Guzman

Friday February 19, 2020

In this assignment, I use a single image to render a scene from multiple viewpoints. The algorithm consists of constructing a 3D representation of the scene, and mapping the image to those surfaces using a unique homography for each surface. A virtual camera can then be moved inside of the 3D representation to view the scene from any angle.

The first step is to calculate the x, y, and z vanishing points from detected lines within the image. I used the python package pylsd-nova to detect the lines, and used RANSAC to filter outliers that did not converge to a given vanishing point. The vanishing points are calculated as follows: The line connecting two points can be expressed as

$$\begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \end{bmatrix} \mathbf{l} = \mathbf{0}$$

where $\mathbf{x}_1^T = [u, v, 1]$ is the homogeneous pixel coordinate of each detected point and $\mathbf{l} = [a, b, c]^T$ defines the line $au + bv + c = 0$. $\mathbf{l}$ exists in the null space of the stacked x1, x2 matrix, so we can solve for it using SVD. The intersection of two lines is then

$$\begin{bmatrix} \mathbf{l}_1^T \\ \mathbf{l}_2^T \end{bmatrix} \mathbf{x} = \mathbf{0}$$

which we can solve similarly. The Z vanishing point is the point that the majority of lines in the image converge to.

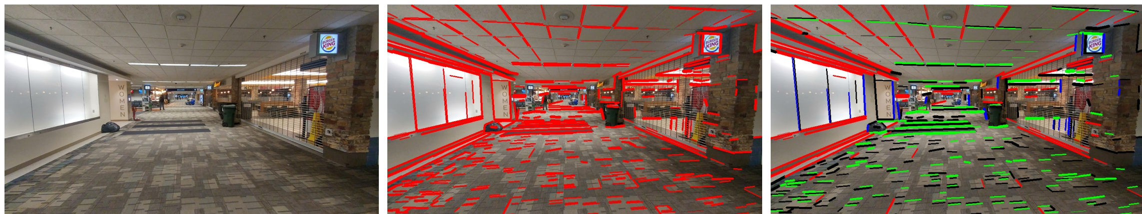

Figure 1: The original image (left), detected lines (middle) and lines after clustering (right).

After finding the Z vanishing point, I cluster the remaining lines into the X and Y directions by comparing their angles. Then, the RANSAC vanishing point calculation is repeated for the X and Y directions. In order to improve numerical stability, I first convert to a matric space by multiplying pixel coordinates by an approximate inverse intrinsic parameter matrix K.

After all vanishing points are known, I can exactly compute the intrinsic parameters K

$$\underbrace{\begin{bmatrix} u_1u_2 + v_1v_2 & u_1 + u_2 & v_1 + v_2 & 1 \\ u_3u_2 + v_3v_2 & u_3 + u_2 & v_3 + v_2 & 1 \\ u_1u_3 + v_1v_3 & u_1 + u_3 & v_1 + v_3 & 1 \end{bmatrix}}_{3\text{x}4} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = 0 \qquad p_x = -\frac{b_2}{b_1}, \quad p_y = -\frac{b_3}{b_1}, \quad f = \sqrt{\frac{b_4}{b_1} - (p_x^2 + p_y^2)}.$$

where u_i and v_i are the pixel coordinates of a given vanishing point.

Using the calculated K matrix, I rectify the image so that all lines in the X and Y directions are aligned with the image plane. This homography is a simple rotation that aligns the vanishing points with the unit axes:

$$R = [p_x, p_y, p_z]$$

$$H = KRK^{-1}$$

$$p_i = K^{-1}u_i/||K^{-1}u_i||$$

Figure 2 shows the result of this transformation. Next, I make a 3D representation of the scene using a box and visualize those points by mapping them back to the image.

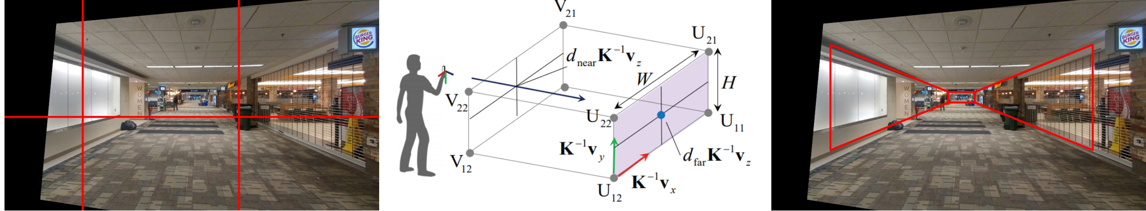

Figure 2: The rectified image (left) and box representation (middle, right).

The last step is to compute the homography that maps each plane to the corresponding locations in the source and target images. This is done in two steps: one homography for mapping the source image to the 3D plane, and another to map the 3D plane to the target image.

Homography mapping from 3D plane to image:

$$\lambda u = K[B_1 \quad B_2 \quad c]\begin{bmatrix}\mu_1\\ \mu_2\\ 1\end{bmatrix} = H\begin{bmatrix}\mu_1\\ \mu_2\\ 1\end{bmatrix}$$

Homography mapping from 3D plane to target image:

$$\lambda\tilde{u} = K[R \quad t]\begin{bmatrix}X\\ Y\\ Z\\ 1\end{bmatrix} = K[R \quad t]\begin{bmatrix}[B_1 \quad B_2 \quad c]\begin{bmatrix}\mu_1\\ \mu_2\\ 1\end{bmatrix}\\ 1\end{bmatrix}$$

$$= K[RB_1 \quad RB_2 \quad Rc+t]\begin{bmatrix}\mu_1\\ \mu_2\\ 1\end{bmatrix} = \tilde{H}\begin{bmatrix}\mu_1\\ \mu_2\\ 1\end{bmatrix} \rightarrow \lambda\tilde{H}^{-1}\tilde{u} = \begin{bmatrix}\mu_1\\ \mu_2\\ 1\end{bmatrix} = \lambda H^{-1}u \rightarrow \lambda\tilde{u} = \tilde{H}H^{-1}u$$

where c is a point on the plane, B_1, B_2 are basis vectors on the plane, and R and t represent the desired camera pose. Here $\tilde{u}$ is the pixel location in the target image and u is the source pixel location. The composed homography $\hat{H} = \tilde{H}H^{-1}$ can then map directly between viewpoints.

Since we have 5 different homographies (one for each plane), we need a mask to filter out points that fall outside the plane. We can check this using the Cheirality condition: $\lambda \geq 0$, $0 \leq \mu_1 \leq \mu_1^{\max}$, $0 \leq \mu_2 \leq \mu_2^{\max}$.

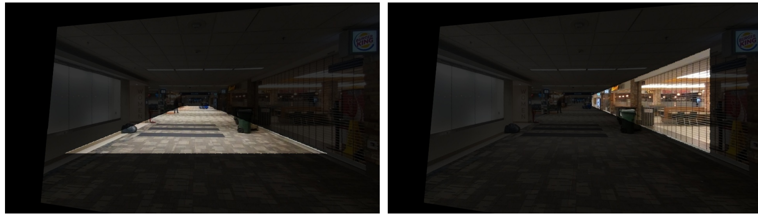

Figure 3: Example masks of the right wall and floor

After we have a valid mask for each plane, we can generate the composite final image by applying the calculated $\hat{H}$ to each image, multiplying by the mask, and summing them. Figure 4 shows the final result for multiple camera positions. Interpolating between camera poses also requires the use of quaternions and spherical interpolation. By doing this process for multiple frames, I generated a fly-through video of the image.

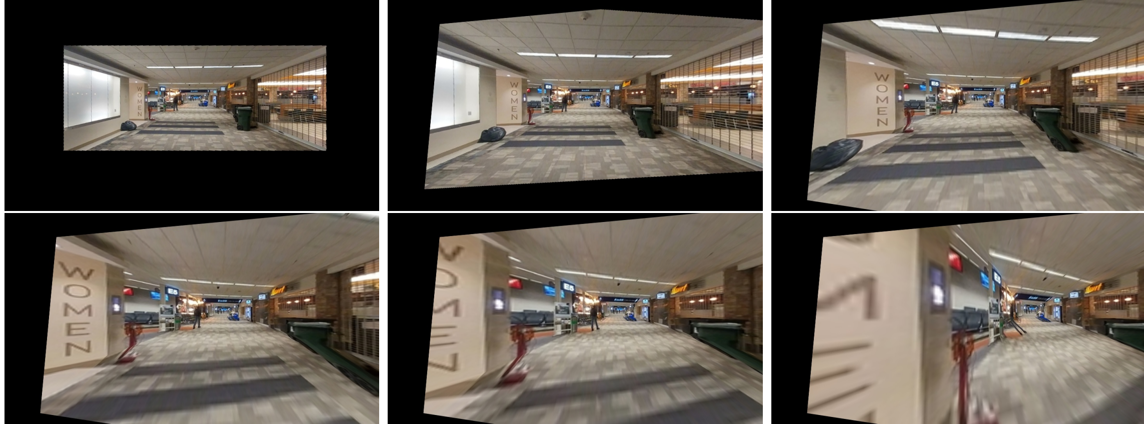

Figure 4: Sample frames of the fly-through. Soon after this, the camera passes through the wall, which could either be an issue with my interpolation or how I calculated the homographies.

Algorithm 1 Single View Image Navigation

1: Load the input image and its line segments.
2: Compute the major z-directional vanishing point and its line segments. ▷ `FindVP`
3: Cluster the rest of line segments into two major directions ▷ `ClusterLines`
4: Compute the x- and y-directional vanishing points. ▷ `FindVP`
5: Calibrate camera intrinsic parameter. ▷ `CalibrateCamera`
6: Compute the rectification homography. ▷ `GetRectificationH`
7: Rectify the input image and vanishing points.
8: Construct 3D representation of the scene using a box model. ▷ `ConstructBox`
9: **for** Each virtual camera pose transition **do**
10:     **for** Each time instant **do**
11:         Interpolate a virtual camera pose between two poses. ▷ `InterpolateCameraPose`
12:         Compute homographies for five planes in the box. ▷ `GetPlaneHomography`
13:         Combine all planes to generate the image.
14:     **end for**
15: **end for**

Figure 5: Tour into image algorithm