CSCI 5563: Assignment #2

Single View Image Navigation

1 Submission

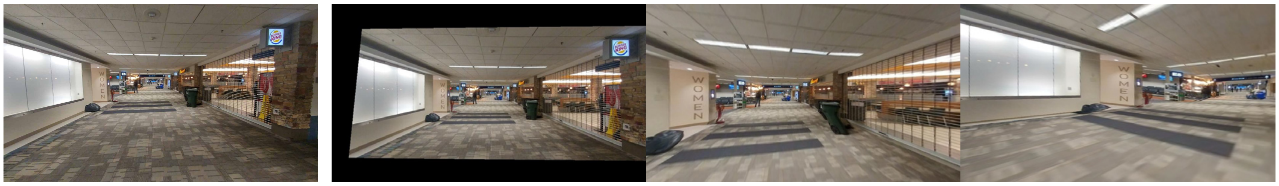
- Assignment due: Feb 19 (11:55pm)
- Skeletal code and data can be downloaded from: `https://www-users.cs.umn.edu/~hspark/csci5563_S2021/hw2.zip`.
- Individual assignment
- Up to 3 page summary write-up with resulting visualization (more than 2 page assignment will be automatically returned.).
- Submission through Canvas.
- Use Python 3
- You will complete HW2.py including the following functions:
  - `FindVP`
  - `ClusterLines`
  - `CalibrateCamera`
  - `GetRectificationH`
  - `ImageWarping`
  - `ConstructBox`
  - `InterpolateCameraPose`
  - `Rotation2Quaternion`
  - `Quaternion2Rotation`
  - `GetPlaneHomography`
- DO NOT SUBMIT THE PROVIDED IMAGE DATA
- The function that does not comply with its specification will NOT BE GRADED.
- You are not allowed to use computer vision related package functions unless explicitly mentioned here. Please consult with TAs if you are not sure about the list of allowed functions.

CSCI 5563: Assignment #2

Single View Image Navigation

2 Overview

In this assignment, you will implement a method to navigate into a photo by reconstructing the scene using vanishing points. The pseudo code can be found in Algorithm 1.



(a) Input image

(b) Output images

Figure 1: Given (a) a single view image, you will generate (b) a sequence of images as if you fly into the scene.

Algorithm 1 Single View Image Navigation

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

3 Line Segment Detection

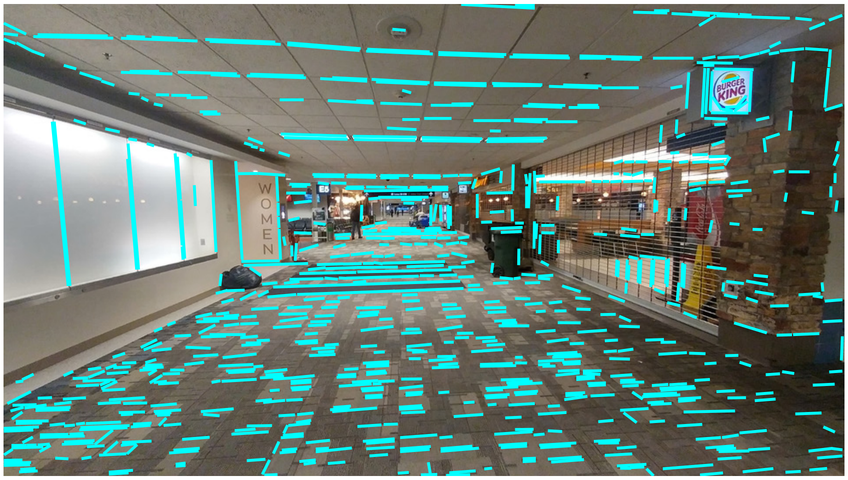


Figure 2: You will use an off-the-shelf software to compute line segments.

Given an image, you will use an off-the-shelf software LSD (`http://www.ipol.im/pub/art/2012/gjmr-lsd/`) to compute line segments. You can install the Python binding of LSD using

```
pip install pylsd-nova
```

The usage of this package can be found at `https://pypi.org/project/pylsd-nova/`.

CSCI 5563: Assignment #2

Single View Image Navigation

4 Vanishing Point Detection

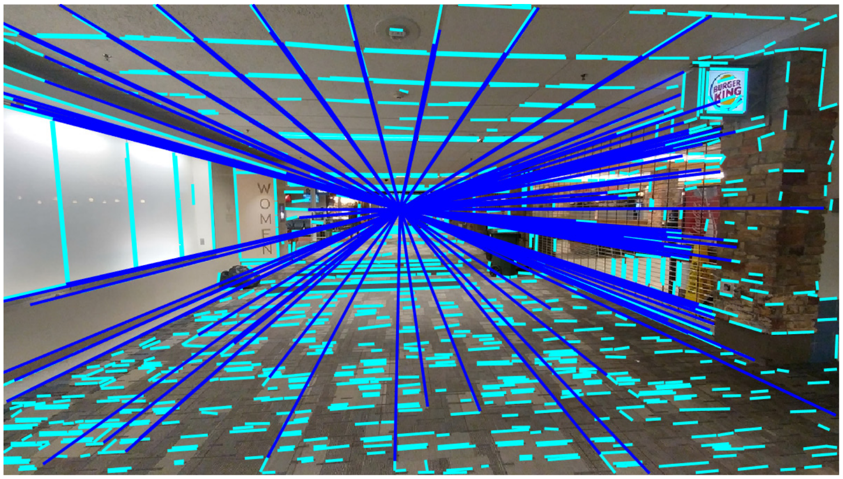


Figure 3: Given the line segments, you can compute the major axis of the scene by finding a vanishing point.

Given the line segments, you will compute a vanishing point as shown in Figure 3. There are many outliers in the line segments that can be robustly filtered out by RANSAC-based estimation. Computing a point at infinity using noisy pixel coordinate is often numerically unstable due to its magnitude. Computation in normalized coordinate (i.e., metric space) can mitigate the numerical instability:

$$\begin{bmatrix}\widetilde{\mathbf{u}} \\ 1\end{bmatrix} = \mathbf{K}^{-1}\begin{bmatrix}\mathbf{u} \\ 1\end{bmatrix}, \tag{1}$$

where $\mathbf{u}$ is pixel coordinate, $\mathbf{K}$ is the intrinsic parameter, and $\widetilde{\mathbf{u}}$ is the normalized coordinate. Since the intrinsic parameter is unknown, we will provide an approximate $\mathbf{K}$ where the principal point is located at the center of image, and the focal length is a half of image width. The accurate $\mathbf{K}$ will be computed by calibration in Section 6. You will need to manually find RANSAC parameters (threshold and the number of iterations). The majority of line segments converge to the z-directional vanishing point, $\mathbf{v}_z$ as shown in Figure 3.

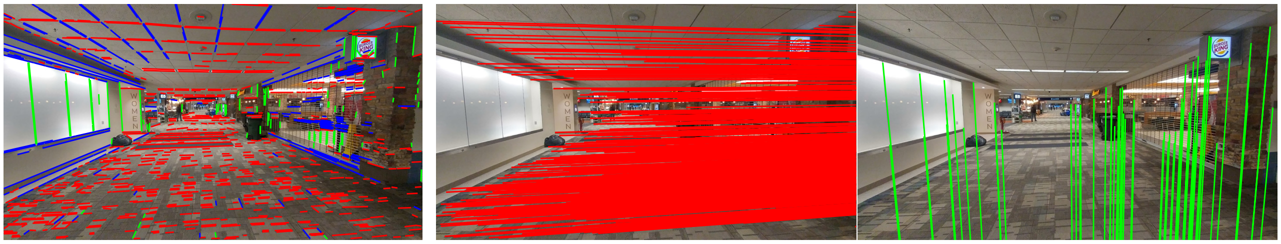
```
def FindVP(lines, K, ransac_thr, ransac_iter):
  ...
  return vp, inlier
```

Input: `lines` $\in \mathbb{R}^{N_l \times 4}$ are a set of line segments where each row contains the coordinates of two points (x_1, y_1, x_2, y_2), i.e., (x_1, y_1) and (x_1, y_1) are the two end points of a line segment, `K` $\in \mathbb{R}^{3\times 3}$ is the camera intrinsic parameter, and `ransac_thr` and `ransac_iter` are the RANSAC threshold (i.e., distance between a line to vanishing point) and the number of iterations. N_l is the number of line segments.

Output: `vp` $\in \mathbb{R}^2$ is the computed vanishing point, and `inlier` $\in \mathbb{Z}^{N_i}$ is the index set

of line segment inliers where N_i is the number of inliers.
Note: The output `vp` must be in pixel coordinate, i.e., you have to de-normalize the coordinate to output the vanishing point in pixel.

5 Other Two Vanishing Point Detection



(a) Line clustering

(b) Other two vanishing points

Figure 4: (a) You will cluster the line segments using heuristics. (b) Within each cluster, you can compute the other two vanishing points.

Computing other two vanishing points $\mathbf{v}_x$ and $\mathbf{v}_y$ is trickier than the first one because the line segments are almost parallel, i.e., the vanishing points form at far beyond the pixel coordinate. This often leads to numerical instability and estimation failure. Instead, you will pre-cluster the line segments using heuristics, e.g., threshold on its slope. Specifically, given a set of line segments excluding the inliers from the first vanishing point detection (blue line segments in Figure 4(a)), you will cluster the line segments into two sets (red and green line segments in Figure 4(a). Given each line segment set, you will use `FindVP` to compute the vanishing point as shown in Figure 4(b). You may need to use different RANSAC parameters from $\mathbf{v}_z$ (threshold and the number of iterations).

```
def ClusterLines(lines):
  ...
  return lines_x, lines_y
```

Input: `lines` $\in \mathbb{R}^{(N_l-N_i)\times 4}$ are a set of line segments excluding the inliers from the first vanishing point detection.
Output: `lines_x` $\in \mathbb{R}^{(N_x)\times 4}$ and `lines_y` $\in \mathbb{R}^{(N_y)\times 4}$ are the set of line segments for horizontal and vertical directions, respectively.

6 Intrinsic Parameter Calibration

Given the vanishing points, you will estimate the accurate intrinsic parameter $\mathbf{K}$. The idea is to enforce orthogonality between X-, Y-, and Z-points at infinity:

$$\mathbf{v}_i^{\mathsf{T}}\mathbf{K}^{-\mathsf{T}}\mathbf{K}^{-1}\mathbf{v}_j = 0 \quad \text{where} \quad i, j = x, y, z \quad \text{and} \quad i \neq j, \tag{2}$$

where $\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z$ are x-, y-, and z-vanishing points in homogeneous coordinate.

Equation (2) can be re-arranged to form a linear system of equations:

$$\mathbf{A}\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \mathbf{0}, \tag{3}$$

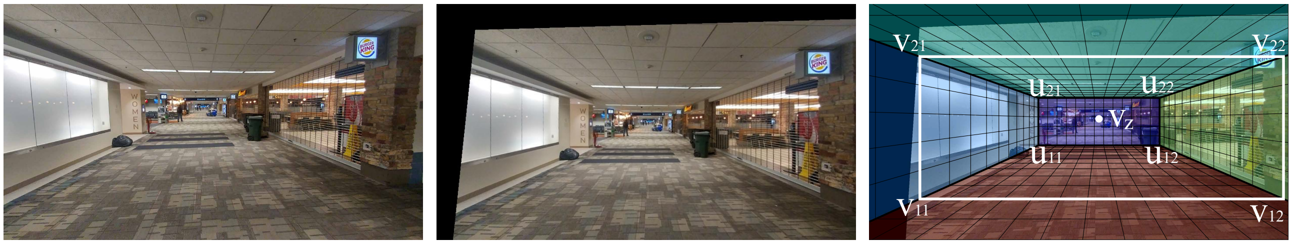
where $\mathbf{A}$ is made of coordinates of the vanishing points, and b_1, b_2, b_3, and b_4 are variables made of the focal length, f and principal point (p_x, p_y). By solving Equation (3), you can derive:

$$p_x = -\frac{b_2}{b_1}, \quad p_y = -\frac{b_3}{b_1}, \quad f = \sqrt{\frac{b_4}{b_1} - (p_x^2 + p_y^2)}. \tag{4}$$

```
def CalibrateCamera(vp_x, vp_y, vp_z):
  ...
  return K
```

Input: `vp_x`, `vp_y`, `vp_z` $\in \mathbb{R}^2$ are the vanishing points in x-, y-, and z-directions, respectively.
Output: `K` $\in \mathbb{R}^{3\times 3}$ is the intrinsic parameter.

7 Scene Rectification



(a) Input image

(b) Rectified image

(c) Line alignment

Figure 5: Using three major axes provided by the detected vanishing points, you will rectify the image.

The detected vanishing points define the major axes of the 3D scene. You will warp your image such that the orientation of the camera is aligned with the major axes of the 3D scene. Specifically, you will find a homography induced by a pure rotation where the rotation can be expressed by the scene's major axes. The vertical lines and horizontal lines in 3D must be vertical and horizontal in 2D as shown in Figure 5(c).

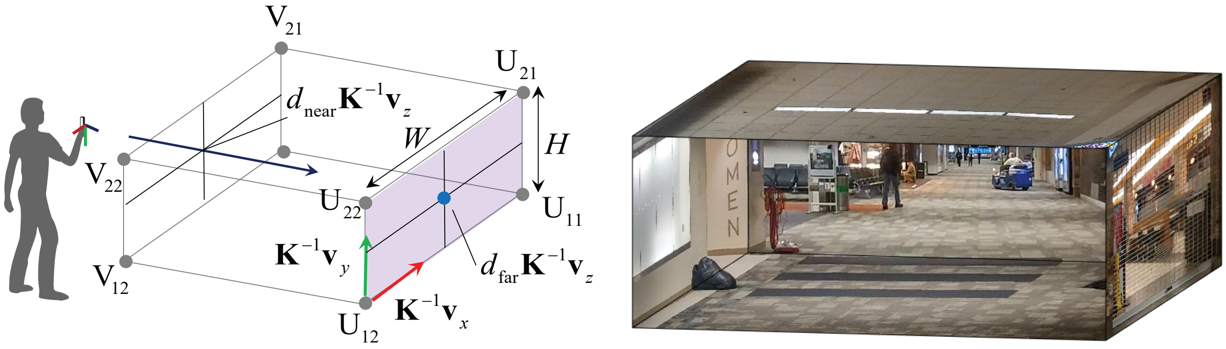
```
def GetRectificationH(K, vp_x, vp_y, vp_z):
  ...
  return H_rect
```

Input: `K` $\in \mathbb{R}^{3\times3}$ is the intrinsic parameter, and `vp_x`, `vp_y`, `vp_z` $\in \mathbb{R}^2$ are the vanishing points in x-, y-, and z-directions, respectively.
Output: `H_rect` $\in \mathbb{R}^{3\times3}$ is the rectification homography induced by the pure rotation.
Note: Make sure the computed rotation has determinant 1.

```
def ImageWarping(im, H):
  ...
  return im_warped
```

Input: `im` $\in [0, 255]^{H_i\times W_i\times3}$ is the image to be warped, and `H` $\in \mathbb{R}^{3\times3}$ is the homography where H_i and W_i are the height and width of the image, respectively.
Output: `im_warped` $\in [0, 255]^{H_i\times W_i\times3}$ is the warped image with the same size as the input image.
Note: You may use `scipy.interpolate` for batch pixel warping with bilinear interpolation

8 Box Scene Representation


V21
U21
$d_{\text{near}}\mathbf{K}^{-1}\mathbf{v}_z$
W
H
V22
U22
U11
$\mathbf{K}^{-1}\mathbf{v}_y$
$d_{\text{far}}\mathbf{K}^{-1}\mathbf{v}_z$
V12
$\mathbf{K}^{-1}\mathbf{v}_x$
U12

(a) Box scene geometry (b) Textured box

Figure 6: Using three major axes provided by the detected vanishing points, you will rectify the image.

The scene geometry can be approximated by a 3D box that contains 5 planes (floor, left/right walls, ceiling, and back). This 3D box is aligned with the x-, y-, and z-directions of the 3D scene. The 3D coordinates of 8 corners of the box can be computed given the far and near depths ($d_{\text{far}}, d_{\text{near}}$), width W, and aspect ratio of the frontal plane, ($a = W/H$) as shown in Figure 6(a). With texture mapping, you can visualize the textured 3D box as shown in Figure 6(b).

```
def ConstructBox(K, vp_x, vp_y, vp_z, W, a, d_near, d_far):
  ...
  return U11, U12, U21, U22, V11, V12, V21, V22
```

Input: `W` is the width of the box, and `a` is the aspect ratio. `d_near` and `d_far` are the depth of the two front and back planes.

Output: `U11`, $\cdots$,`V22` $\in \mathbb{R}^3$ are 8 corners of the box defined in Figure 6(a).

Note: Due to the error in the intrinsic parameter and vanishing points, the major axes defined by vanishing points may not be orthogonal. To ensure orthogonality, you can apply Gram–Schmidt process to rectify the x- and y-directions given z-direction.

9 Camera Pose Interpolation

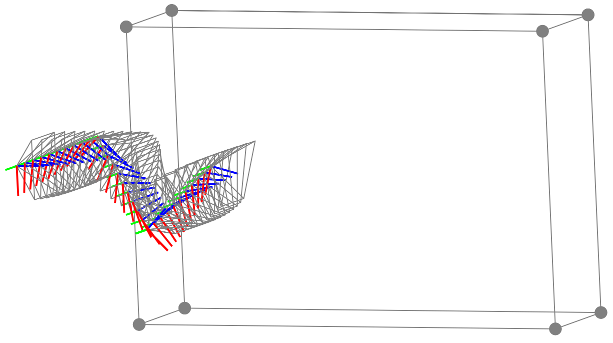


Figure 7: Smooth transition of images can be achieved by camera interpolation.

To generate a smooth transition between two end-point camera poses, you will interpolate the pose. Four poses are linearly interpolated in Figure 7.

```
def InterpolateCameraPose(R1, C1, R2, C2, w):
  ...
  return Ri, Ci
```

Input: $\texttt{R1}, \texttt{R2} \in \mathbb{R}^{3\times3}$ are two camera rotation matrices, and $\texttt{C1}, \texttt{C2} \in \mathbb{R}^3$ are two camera optical centers to be interpolated. $w \in [0, 1]$ is the weight between two poses, i.e., the interpolated pose is $(\texttt{R1}, \texttt{C1})$ at $w = 0$ and $(\texttt{R2}, \texttt{C2})$ at $w = 1$.
Output: $\texttt{Ri} \in \mathbb{R}^{3\times3}$ and $\texttt{Ci} \in \mathbb{R}^3$ are the interpolated camera pose.
Note: You will use a linear interpolation for the camera optical center and spherical linear interpolation for the camera rotation. You may need to implement `Rotation2Quaternion` and `Quaternion2Rotation` (you are not allowed to use a built-in function for rotation/quaternion conversion.).

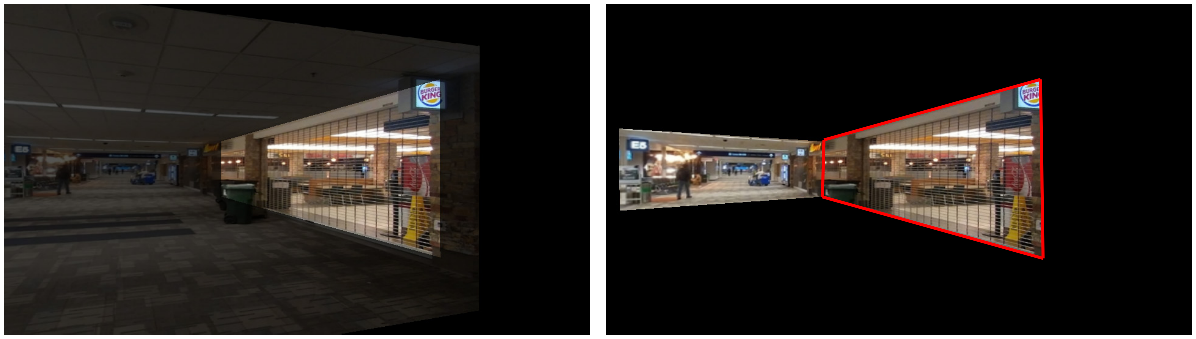
```
def Rotation2Quaternion(R):
  ...
  return q
```

Input: $\texttt{R} \in \mathbb{R}^{3\times3}$ is a rotation matrix.
Output: $\texttt{q} \in \mathbb{R}^4$ is the unit quaternion (w, x, y, z) representing the rotation.

```
def Quaternion2Rotation(q):
  ...
  return R
```

Input: $\texttt{q} \in \mathbb{R}^4$ is a unit quaternion (w, x, y, z).
Output: $\texttt{R} \in \mathbb{R}^{3\times3}$ is the rotation matrix.

10 Plane Mapping and Image Composition



(a) Texture mapping for right wall

(b) Image composition

Figure 8: You will map the texture of each face of the box to the canvas.

You will generate a composite image in a canvas by mapping the texture from the input image. This mapping is induced by the plane homography, i.e., each face of the box. This mapping can be derived by a composition of two homographies:

$$\lambda \begin{bmatrix} \mathbf{u} \\ 1 \end{bmatrix} = \widehat{\mathbf{H}} \begin{bmatrix} \boldsymbol{\mu} \\ 1 \end{bmatrix}, \qquad \lambda \begin{bmatrix} \widetilde{\mathbf{u}} \\ 1 \end{bmatrix} = \widetilde{\mathbf{H}} \begin{bmatrix} \boldsymbol{\mu} \\ 1 \end{bmatrix} \tag{5}$$

where $\boldsymbol{\mu} \in \mathbb{R}^2$ is the 2D parametrization of a 3D point on the plane, i.e., $\mathbf{X} = \mu_1 \mathbf{b}_1 + \mu_2 \mathbf{b}_2 + \mathbf{o}$ where $\mathbf{b}_1, \mathbf{b}_2 \in \mathbb{R}^3$ are two orthogonal vectors in the plane (perpendicular to the surface normal), and $\mathbf{o} \in \mathbb{R}^3$ is a point of the plane. $\mathbf{u}$ is the projection of the 3D point onto the rectified camera, and $\widetilde{\mathbf{u}}$ is the projection of the 3D point onto the canvas (i.e., a new camera pose). You will transport pixels in the rectified image to the canvas, i.e., $\lambda \begin{bmatrix} \widetilde{\mathbf{u}} \\ 1 \end{bmatrix} = \mathbf{H} \begin{bmatrix} \mathbf{u} \\ 1 \end{bmatrix}$.

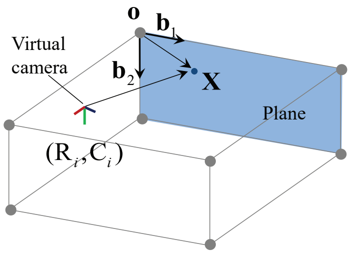


Figure 9: Plane and virtual camera geometry.

There are five planes to map, independently, and therefore, for each pixel in the canvas, you have to reason about which plane it belongs to. A 3D point that maps texture must meet the following two conditions:

- The point and virtual camera must satisfy cheirality, i.e., the point behind the virtual camera pose is not visible.
- The point must lie in the plane in which range is defined by the four corners of the box face.

```
def GetPlaneHomography(p11, p12, p21, K, R, C, vx, vy):
  ...
  return H, visibility_mask
```

Input: $\texttt{p11}, \texttt{p12}, \texttt{p21} \in \mathbb{R}^3$ are top-left, top-right, and bottom-left corner coordinates of a face in the 3D box, respectively. $(\texttt{K}, \texttt{R}, \texttt{C})$ are camera parameters, i.e., the projection matrix can be made by $\texttt{KR}\begin{bmatrix}\mathbf{I}_3 & -\texttt{C}\end{bmatrix}$ where $\mathbf{I}_3$ is the 3×3 identity matrix. $\texttt{vx}, \texttt{vy} \in \mathbb{R}^{h \times w}$ specifies all x and y coordinates in an image where h and w are height and width of the canvas image.

Output: $\texttt{H} \in \mathbb{R}^{3\times 3}$ is the homography that maps the rectified image to the canvas, and $\texttt{visibility\_mask} \in \{0, 1\}^{h\times w}$ is a binary mask indicating membership to the plane constructed by `p11`, `p12`, and `p21`.

Note: Figure 8(a) illustrates the warped image and its mask. Brighter pixels indicates the pixels that belong to the right face in the box, which are mapped to generate the texture in the canvas in Figure 8(b).