Structured Semantic 3D Reconstruction (S23DR) Challenge

Maksym Ivashechkin
University of Surrey
Guildford, United Kingdom
m.ivashechkin@surrey.ac.uk

Abstract

The S23DR challenge aims to tackle a 3D wire-frame reconstruction problem from a set of images with semantic segmentation. Limited prior knowledge and noisy input create the main challenges for solving this task. This work discusses a possible solution that ranked fourth in the private set evaluation. The main contributions are about improving 2D vertex and edge detection, estimating 3D vertices by ray-casting the 2D observations, and incremental vertex merging.

Step-by-step solution:

1. The original 'hand-crafted' solution has an effective method for extracting apex and eave 2D vertices from a segmented image. However, the threshold for `cv2.inRange` was set too low, causing many potential vertices to be missed. By increasing the threshold, the method was able to reliably identify a sufficient set of apex and eave 2D points.
2. The original method for edge extraction was not working very well. The problem was again due to a low threshold and the noisiness of the segmentation. The edge detector was returning a lot of small edges that should have been merged. Also, the original solution was returning many repetitive edges and vertices. Therefore, I updated the edge detection process, see also Fig. 1:
   (a) I have replaced OpenCV functions for edge masking:
   ```
   morphologyEx(inRange(gest_seg, e_color-1.5, e_color+1.5), MORPH_DILATE, np.ones((11, 11)))
   ```
   with a low-level solution that gives better results:
      i. at first I found a binary mask:
      ```
      logical_and((gest_seg > e_color-10).sum(-1) == 3, (gest_seg < e_color+10).sum(-1) == 3)
      ```
      ii. and then I ran convolution to smooth the initial mask:
      ```
      conv2d(mask2.reshape(1,1,*mask2.shape), torch.ones(KS, KS)/KS**2, padding='same')
      ```
   (b) To further mitigate the issue with small edges, I have added an algorithm that merges two edges if their endpoints are too close.
   (c) Finally, I have replaced the original method of matching apex/eave vertices to edges with my own approach. I have noticed that some apex/eave points are missing from the initial detection step but are present when considering edge endpoints. Therefore, I wanted to add them during the process. For each edge, I consider four different scenarios:
      i. If the edge is connected to two apex points (i.e., the distance is within a threshold), add both points.
      ii. If the edge is long enough and connected to one apex point on the left, add the right edge point to the new vertices.
      iii. If the edge is long enough and connected to one apex point on the right, add the left edge point to the new vertices.
      iv. If the edge is long enough and not connected to any apex points, add both edge endpoints.
3. Given a set of 2D vertices, the main task is to uplift them to 3D space. I did not want to use an estimated depth map because they did not appear accurate. Instead, I utilized a ray-casting technique. For each 2D vertex, I parameterized a ray using the provided extrinsic and intrinsic camera parameters.
   For each 2D vertex $\mathbf{x} = [u, v, 1]$, the origin of the ray is the camera center $\mathbf{C}$, and the ray direction is $\mathbf{d} = \mathbf{R}\mathbf{K}^{-1}\mathbf{x}$, where $\mathbf{R}$ is the camera-to-world rotation matrix, and $\mathbf{K}$ is the intrinsic matrix. The goal is to find the intersection of a ray $(\mathbf{C}, \mathbf{d})$ with the COLMAP 3D points.
   This involves finding the projection of a 3D point onto the ray. By performing batch calculations, I determined the

distances from all rays (derived from the 2D vertices) to all 3D COLMAP points. I then selected the projections on the ray that had the minimum distance to the COLMAP points. Given that COLMAP points are not precise, I decided to use the 3D points on the ray as the final result.

(a) `colmap_ray_dir = t_inv[None,None].repeat(len(uv),0) - points3d.T[None] # UV x COLMAP x 3`
(b) `proj_scale = -(colmap_ray_dir * rays_d[:,None]).sum(-1) / (rays_d * rays_d).sum(-1)[:,None] # UV x COLMAP`
(c) `closest_pts_on_rays_to_all =t_inv[None,None] + rays_d[:,None] * proj_scale[...,None] # UV x COLMAP x 3`
(d) `actual_dists = np.linalg.norm(closest_pts_on_rays_to_all - points3d.T[None], axis=-1) # UV x COLMAP`
(e) `min_dists_idxs = actual_dists.argmin(-1) # UV`
(f) `closest_pts_on_rays = closest_pts_on_rays_to_all[np.arange(len(uv)), min_dists_idxs] # UV x 3`

4. I have also modified the algorithm for merging vertices. I rewrote the code for merging and pruning, making the entire process incremental. In other words, I start merging with a small threshold and incrementally increase it until either the maximum threshold is reached or the number of merged vertices satisfies the desired maximum number of vertices in my solution.
5. I noticed that returning any edge (even a single one, whether the longest or the median) only increases the error metric. Therefore, in the end, I decided not to return any edge. However, edges were still used throughout the method to keep track of connected vertices, especially during the vertex merging process.

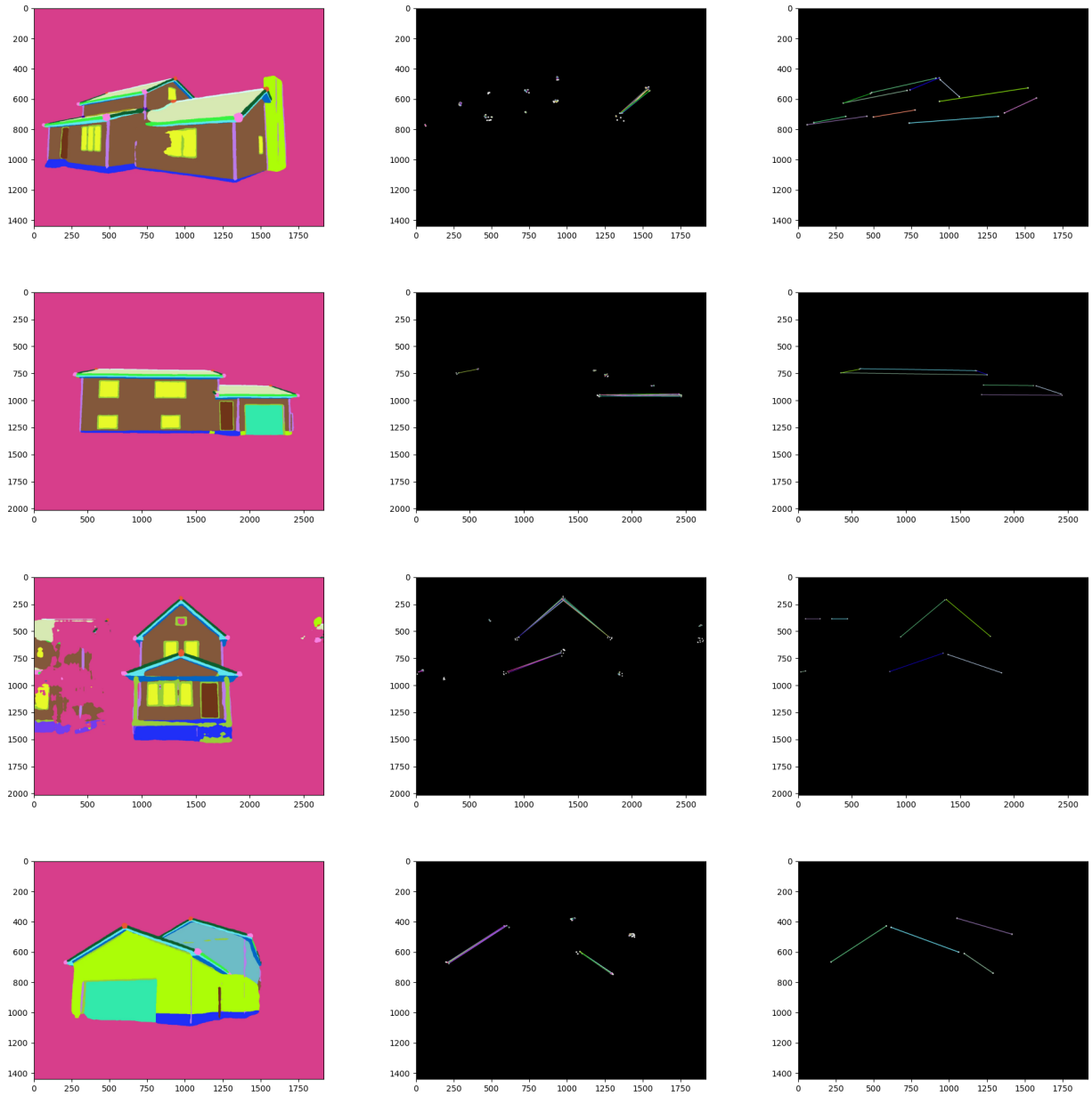

Figure 1. From left to right: original segmentation – baseline detection (even with increased threshold level from 0.5 to 1.5) – the proposed detection. This figure demonstrates more accurate results obtained with the proposed detection method compared to the baseline approach (in the middle). The proposed detection returns more edges and vertices that do not repeat.