S23DR write-up

Serhii Ivanov
Ukrainian Catholic University
Lviv, Ukraine
serhii.ivanov@ucu.edu.ua

Abstract

This write-up explores a solution to wireframe construction problem.

1. Introduction

This is a write-up for the S23DR competition. The model is an improved version of the handcrafted solution. The main components are 2D vertex and edge detection and depth estimation. Edge detection consists of basic connected components analysis with additional checks for connection validity. In this solution the point cloud is used to estimate depth using the mean depth of N closest projected points.

2. The method

2.1. Vertex Detection

My solution extracts the vertex mask for each vertex type from the “gestalt” segmentation. Series of dilations and erosions clean the mask and connected components analysis helps find the centroids and radii of the vertices.

2.2. Line Detection

The algorithm preprocesses the line mask almost in the same way as in vertex detection. Some classes (such as “ridge”) require additional dilations or bigger kernel sizes to be noticed by the next algorithm.

The next step involves Hough line detection, which returns the line ends. There is a special parameter that expands the lines along the direction on both ends. This is due to the requirement in the line agglomeration stage for the line ends to be in range of the corresponding vertices. This way there is a bigger chance that any given line intersects the corresponding vertex, but it also marginally increases the number of false positives.

The best solution expands the lines only once. See why in the metric discussion.

2.3. Missing Vertex Detection

The “gestalt” images have missing vertices, which are extremely easy to detect. Usually they occur where a ridge and a rake form an apex. So the solution does exactly that. It checks if the ridge and rake ends are close, and then returns the inferred vertices. In order to avoid detection of existing vertices the algorithm checks if any other points are in range. Albeit, it does not do so with the points it inferred, which now I realize is a mistake, which led to more false positives due to multiple lines per the same edge.

The issue of missing vertices exists between hips and ridges too, but is ignored by the solution. See why in the metric discussion.

2.4. Line Agglomeration

Line agglomeration begins when all possible lines are detected. It assigns a corresponding vertex in range of a radius for each line end point. Basically, it fits arbitrary lines onto a predefined wireframe. Edges that do not fulfill the vertex requirements are filtered out. The next step looks into the found lines and checks if the found line aligns with the fixed line within some threshold of degrees.

Both radius and degree solutions derive from the nature of 2D projections.

The vertex size correlates with the distance to the camera and thus is a valid option when trying to restrict the line connections. The use of equal radius would mangle the two lines that ended far in the distance and had vertices near each other. KDTree is an extremely useful data structure here. It has a convenient interface for querying points in a radius and smaller computational intensity compared to finding distance between each element of two sets.

The degree solution arises from the ambiguity of vertex radius on 2D projection and noise in the line detection which occurs because of the thickness of the mask (there is a substantial amount of lines detected) and line extension. For example, a line that only partly touches the vertex circle on its edge is the wrong one. It should pass through the vertex and not be partly in orbit. The benefits of this solution are clearly visible when the vertex radii are big.

2.5. The Depth Estimation

The estimated monocular depth proved to be completely inaccurate. Each depth image was either warped, required some sort of coefficient, had inaccurate camera parameters or was filled with noise on important key points.

Thus I had to reject this data and move on with a different approach.

The solution uses projection of the point cloud for depth estimation.

Initially it performs denoising and unwanted object removal on the point cloud using the DBSCAN algorithm. With the preprocessing done point projection takes place which takes note of the projected point depth. In order to combat its sparsity, a variation of nearest neighbor interpolation, that rejects neighbors beyond a certain range and outputs a value that is a mean of N neighbors, estimates the found vertex depth. With that information the algorithm proceeds to transform the vertex coordinates into world coordinates. Lastly, it merges the vertices, removes vertices and edges with nans and returns the output to the client.

Since not every image has key point information, the monocular depth map acts as a backup and is used as in the handcrafted solution.

The trouble with this approach is that point clouds are almost completely missing an important component – roofs. I failed to fix this issue.

The first approach was to use the depth maps in combination with the interpolator. If queried vertices are close to the projected point then its value is selected, else value from the depth map is used.

The second approach was to find how to shift and scale the depth map so it fits the point cloud with the least amount of error, but swiftly discarded due to bad visual results which included inability to unwarp the depth map.

The third approach was to train an FCNN that would learn how to add the depth maps generated by projecting the points and monocular ones. The projection of 3D ground truth mesh was used as the target. This approach yielded a highly unstable model with dubious accuracy.

3. Discussion

3.1. The results

I finished fifth with the final score of 2.0144 on the private dataset. Despite the handcrafted solution’s simplicity, my solution managed to prove that there is a lot of room for improvement for this approach.

3.2. Area of improvement

Since the great weakness of the algorithm is the correct depth estimation, a more robust algorithm should be used instead, coming up with which would require additional research and expertise.

3.3. Validation Pipeline

It was a great pleasure and a great curse that the solution’s performance was the same on validation data as on the test data. On one hand I could quickly validate the solution and be confident about the results. On the other hand, it led me down the spiral of constant fine-tuning that prevented me from exploring actual solutions.

3.4. My mistakes

My major error was that I had done no research on wireframe construction and tried to do it all on my own.

3.5. Thoughts on the metric

I have no prejudice against the metric, but it is unfortunate that the improvement of vertex detection worsened the solution's accuracy (by about 0.2). The explanation to the line detection and missing vertex detection is that additional vertices and edges helped accumulate the error that was generated by inaccurate depth detection.

References

[1] Langerman, Jack and Korkmaz, Caner and Chen, Hanzhi and Gao, Daoyi and Demir, Ilke and Mishkin, Dmytro and Birdal, Tolga, S23DR Competition at 1st Workshop on Urban Scene Modeling @ CVPR 2024, usm3d.github.io, 2024