Appendix A: TSDF generation

As described in section 4.2. our goal is to find a transformation from the triangle space to the TSDF space. For that we explained in algorithm 1. how we calculate the closest distance, from a point in space to each triangle. However, doing this for each voxel in the TSDF volume is too computational expensive as this would require for each of the $512^3 = 134217728$ voxels to calculate the distance to each polygon, where there could be per scene several thousands to millions. To achieve this several steps have to be done, which are described next.

Preprocessing

As explained in section 4.1 we transform our polygons into the camera frustum, we then clip all polygons at the near and far plane and remove all polygons, which are not in the camera frustum plus a boundary of b. This b is necessary to keep the distance calculation on the boundaries consistent. A b twice the σ_{tsdf} is sufficient, to ensure that all voxels on the boundaries have the correct value. In the final step we filter out all polygons, which surface is nearly zero. As these are usually artifacts from the projection and cause problems during the distance calculation.

TSDF calculation

As computing the exact distance to the closest polygon for each voxel takes too much time, we approximate this distance by using the fact that neighbouring voxels often have the same closest polygon. We therefore do not loop for each voxel over all possible polygons to find the closest one, but instead collect the closest polygons of the already visited neighbouring voxels and then only go through this small collection of polygons.

In more detail, the algorithm works as follows: We first efficiently look for intersections between all voxels and polygons using an octree. Now we begin a flood fill algorithm starting with the voxels that do intersect with at least one

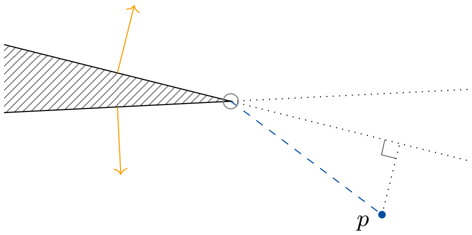

Fig. 1. Two polygons in a top view, which share the circled edge. The distance to this edge in blue shows the distance to the closest point on both polygons. The area in gray shows where the object is filled.

polygon. Every time a new voxel is visited, we first collect all the closest polygons of all already visited voxels in the 5x5x5 area centered around the current voxel. If hereby one of the voxels has intersecting polygons, then these are added to the collection, too. Now based on this collection the closest polygon and its distance to the current voxel is computed. The distance is clipped at $[-\sigma_{tsdf}, \sigma_{tsdf}]$.

To further improve efficiency, the distance can be directly determined as $\pm\sigma_{tsdf}$, if all visited neighbouring voxels have their distance set to $\pm\sigma_{tsdf}$.

On top of the distance, we need to determine for each voxel, if it is inside of an object or outside, to assign the right sign to each distance value. For that we use the normal of the polygon as described in algorithm 1.

There is only one corner case, were this does not work. If the closest point is on the edge of a polygon it is usually the closest point to an other polygon, too. It might be now that this point lies below one polygon but is still outside of the object, as can be seen in figure 1. Here, the upper polygon has the closest distance at the circled point, which depicts the edge of the triangle. This distance between the blue point p and the edge is shown as a blue dashed line. But, based on the normal of the upper triangle, it would be inside of the object, to avoid this we walk over all polygons, which have the same closest distance. If one of them says the point is located outside the object, we mark the voxel as outside and give it a positive sign.

Postprocessing

As we are working mainly with the SUNCG dataset, where not all objects consist of only an outer hull, it is necessary to clean the resulting TSDF volume. Several objects like walls have double surfaces or cupboards, which have backsides to the doors. This causes voxel cells which are close to such inside polygons to be wrongly classified as outside, although they are actually contained in a volume. As this would taint our training data, we had to find a way of removing such artifacts to only obtain the outer shell of all objects in the scene.

We propose a several step solution to find the free and occupied area. At first, we mark all voxel with negative signs as inside voxels and select all surrounding voxels as boundaries. Then, we start from the beginning of the camera frustum and walk with a flood-filling algorithm over all still unmarked voxels and declare them as free voxels. All voxels, which have not been reached that way are declared to be inside of an object. Finally, we check for each border voxel, with a Dijkstra algorithm if an outside voxel can be found in a certain amount of steps. If that's the case we mark it as outside else as an inside voxel.

Now all signs are correct, but the distance values are not. In order to correct this, we walk over all inside voxels and calculate the distance based on the neighboring outside voxel. By using a flood fill algorithm, we can then propagate this distance into the filled space.

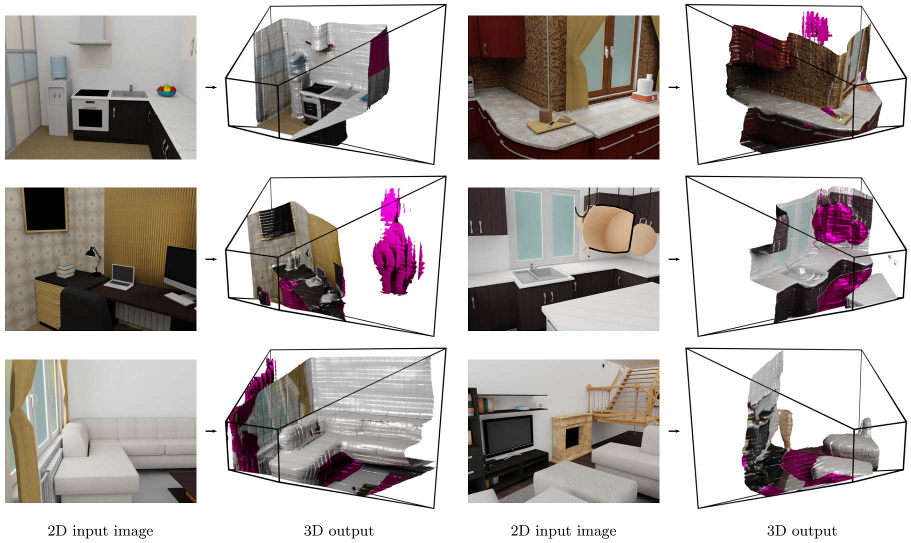

Fig. 2. A few results on the SUNCG dataset. Each of them with the color image and the corresponding reconstruction.

Appendix B: More quantitative results

We provide here some additional results on the SUNCG dataset. Some images have interesting features, which we are going to highlight next. For example in the second image on the left, the chair in the front and the thin table were successfully reconstructed. The network must have understood that this black object in the front is a chair and has assigned a proper occupation depth. Nonetheless, has the background behind the wall some artifacts, these originate from the low value used in the loss shaping behind the walls. Adjusting this might resolve this, but could also cause other problems.

In the lower right do we show a scene, which has a collection of small couch elements in the image. The two big ones on the right side in the image are properly reconstructed and also the start of the fire place as well. But, in the left lower part of the image do we see a corner of a couch, which was omitted in the final reconstruction. Sometimes, the network favors overall structure over small details. In order to improve this our idea is to improve the structure of smaller things in the latent code, because the autoencoder already is able to encode them, but the reconstruction network struggles to reconstruct them.

All of these ideas are for future work.