

A Additional Visualizations for the HOUSELAYOUT3D Dataset

Fig. 9 visualizes the scenes in the annotated dataset, while Fig. 8 shows a screenshot of PinPoint [20], the tool used to create the annotations.

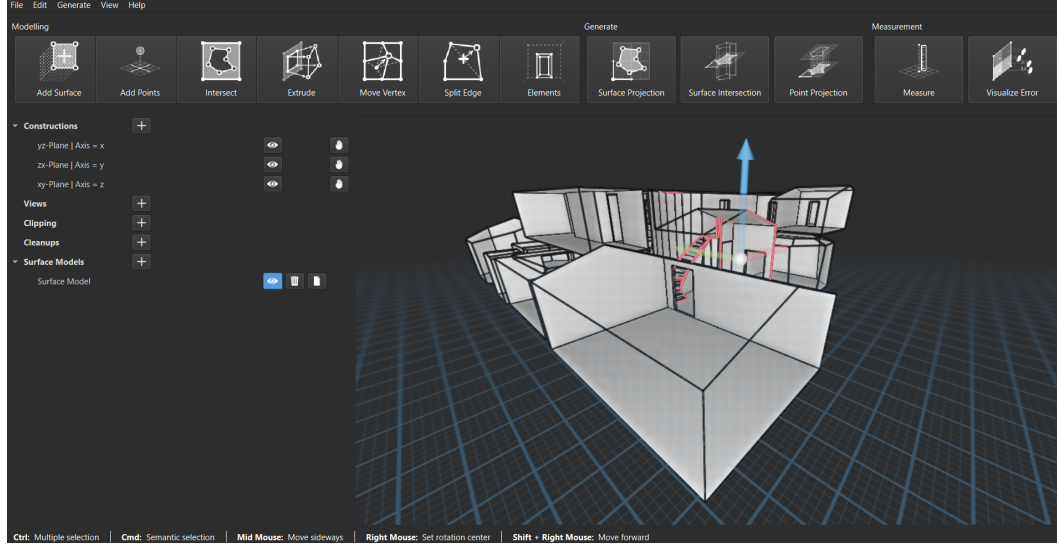


Figure 8: Screenshot of PinPoint [20], the layout annotation tool used to annotate the scenes.

B Re-Classification of COCO [27] Classes

In Sec. 4.2, we segment the input images using OneFormer [26]. It segments the images into COCO [27] classes, which we re-classify into four semantic classes of interest. Table 5 summarizes this re-classification. For window detection in Sec. 4.4, we make two changes to this classification. Firstly, we do not consider mirrors windows. Secondly, we add the surface classes *window blind* and *curtain* to the window classes. Furthermore, the surface class is maintained in the layout skeleton and the prototype layout (but not in the output).

Semantic Class	Category	COCO [27] Classes
Structure	Wall	wall-brick, wall-stone, wall-tile, wall-wood, wall-other-merged
	Ceiling	ceiling-merged
	Floor	floor-wood, floor-other-merged, rug-merged
	Surfaces	cabinet-merged, door-stuff, curtain, window-blind
Geometrically inaccurate surfaces	Windows	window-other
	Mirrors	mirror-stuff
	Outdoor/Noise	gravel, tree-merged, sky-other-merged, pavement-merged, grass-merged, dirt-merged
Stairs	Stairs	stairs
Objects	Object	Rest

Table 5: Mapping of COCO [27] classes into four semantic classes. The *structure* class is used to construct the layout skeleton; the *geometrically inaccurate surfaces* are removed; the *objects* are used to fill holes, and the *stairs* are added back once the scene graph is created (Sec. 4.4). During prototype fitting (Sec. 4.3), we further distinguish between walls, ceilings, floors, and generic surfaces among the structures. Unlike mirrors, the outdoor classes are also used for window detection in Sec. 4.4 because they are typically visible through windows in indoor environments.

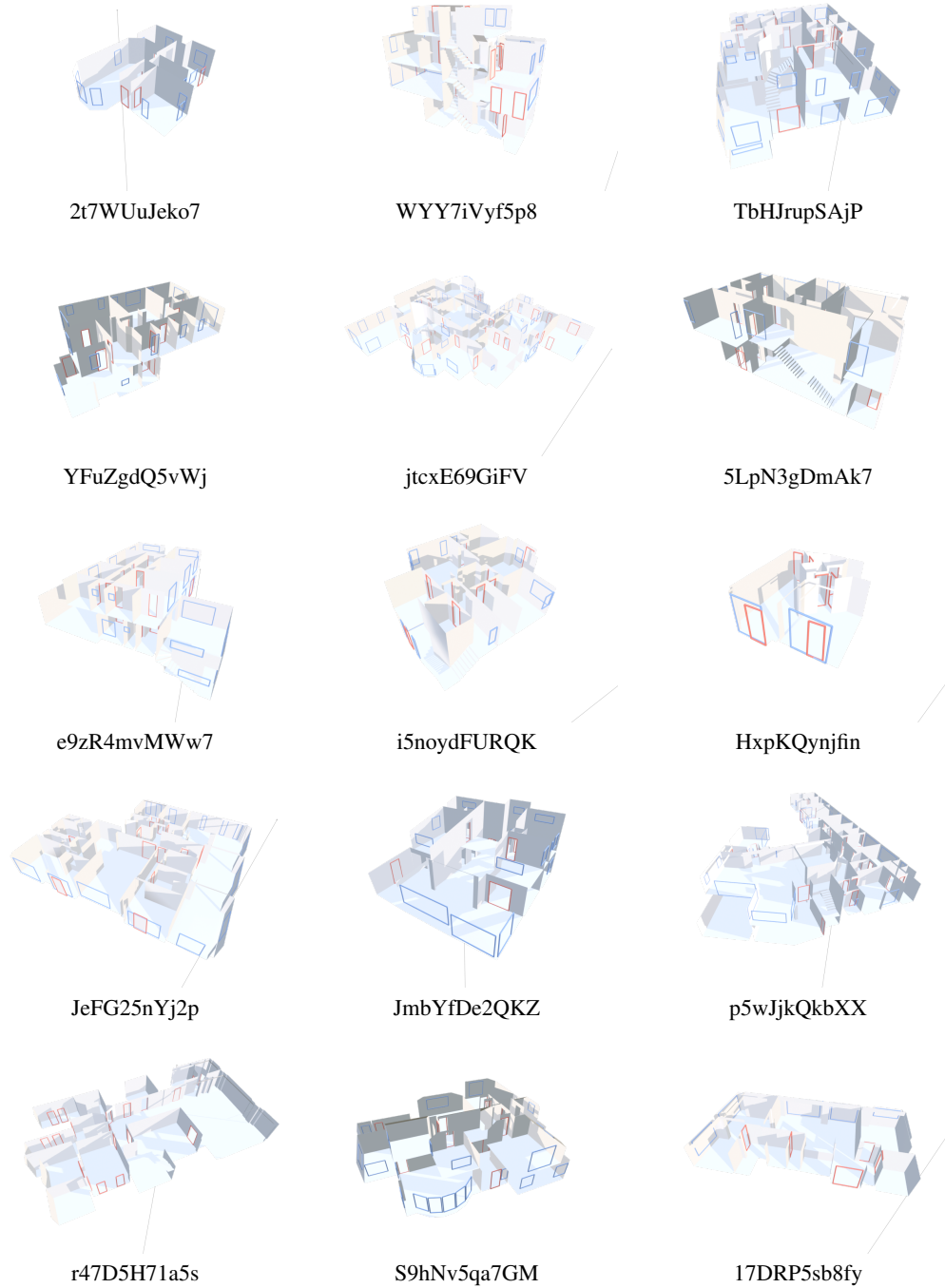


Figure 9: Visualizations and scene names of the scenes in the HOUSELAYOUT3D dataset. (The scene in Fig 1 was omitted).

C Implementation details for Layout Prototype Fitting

C.1 Initialization of a Polygon Set from the Layout Skeleton

In Sec. 4.3 we fit a set of polygons to the layout skeleton to produce the *layout prototype*. Specifically, we sequentially fit one or more planes to each superpoint in the clustering using model fitting by random sample consensus (RANSAC [?]). Then we extract polygons from the connected components of plane inliers (*i.e.* the points close to the planes): that is, we use the connectivity of the skeleton

Require: A mesh M segmented into clusters $S = \{S_1, S_2, \dots, S_n\}$

Ensure: A set of planar 3D polygons \mathcal{P}

```
1: Mark every vertex in  $M$  as unassigned
2: Initialize  $\mathcal{P} \leftarrow \emptyset$ 
3: while there is a cluster with more than  $K$  unassigned vertices do
4:   Choose the cluster  $S^*$  with the most unassigned vertices
5:   Fit a plane to  $S^*$  using RANSAC
6:   Find all unassigned vertices in  $M$  that lie close to the plane (the inliers)
7:   Assign the vertices of the connected component of inliers that overlaps the most with  $S^*$  to a
   new plane (connected w.r.t the mesh edges).
8:   Extract polygon  $P$  from the boundary of the triangles of the connected component, and add it
   to  $\mathcal{P}$ 
9: end while
10:
11: return  $\mathcal{P}$ 
```

Algorithm 1: Initialization of 3D Polygon Set. We extract a set of planar 3D polygons from the clustered *layout skeleton* mesh by sequentially fitting planes to the superpoint clusters obtained in Sec. 4.2.

891 mesh to extract connected components of plane inliers. Then we take the boundary of each connected
892 component as a polygon. Algorithm 1 describes the procedure.

893 C.2 Implementation of a 3D Planar Polygon Set

894 In Sec. 4.3 we fit a set of 3D polygons to the vertices of the layout skeleton mesh using gradient
895 descent.

896 For a set of N polygons and V vertices, we optimize the following parameters: 1) plane equations of
897 shape $(N, 4)$ and 2) the vertex positions of shape $(V, 3)$. The parameters' gradients are computed
898 using backpropagation.

899 **Implementation of 3D Planar Polygons as Triangle Meshes** We build our implementation of
900 a 3D polygon set on pytorch3d triangle meshes [36]. That is, we triangulate each polygon into
901 triangular faces using Constrained Delaunay Triangulations [30] (CDT). To ensure that the polygons
902 are planar, we for each polygon maintain a trainable plane equation. Upon accessing the (trainable)
903 3D position of a vertex, we first project the vertex to the plane constraint of the polygon it belongs to.
904 Periodically, we update the original vertex position with the projected (constrained) position to avoid
905 strong drift in the original vertex positions.

906 **Vertex Sharing** We allow and encourage polygons to share vertices. If two vertices of different
907 polygons are merged, this implies that we require the vertex to satisfy two plane equations. In that
908 case, we project the vertex to the intersection line between the two polygons upon accessing its
909 position.

910 Generally speaking, we store for each vertex up to three plane constraints based on the polygons it is
911 part of, and project it to the closest point satisfying the constraints upon accessing its position.

912 To avoid training instabilities, we avoid merging vertices of near-parallel polygons.

913 **Re-Triangulation** Upon adding triangle faces to polygons (projecting objects to the floor) or
914 merging polygons, we re-triangulate the surface of each affected polygon using a CDT.

915 D Creation of a Scene Graph of 2D Floorplans: Detailed Description

916 In this step we use the prototype layout and its semantics to (1) identify the different levels (floors) of
917 the building, (2) create a 2D layout (floorplan) of each level, and (3) segment each level into rooms,
918 extracting a per-level 2D scene graph from each floor and (4) detect stairs to connect the individual
919 levels. At a high level, the process can be summarized as follows:

920 **To identify building floors,** we use the floor-classified polygons of the layout prototype, merging
921 close levels with similar heights.

922 **To create a 2D floorplan of each level**, we merge each level’s floor polygon(s) with suitable ceiling
923 polygons - since ceilings are rarely occluded by objects and thus are more robustly represented in the
924 prototype layout.

925 **To segment each level into rooms** we apply Hov-SG [15]’s room segmentation algorithm on each
926 2D floorplan (and the walls of the prototype layout). The segmentation outputs a scene graph with
927 rooms as nodes, and *openings* as edges. We consider an opening edge a *door* if its width is below
928 1.5m. Otherwise, we retain its edge but label it as *opening*. Furthermore, each room is associated
929 with a room type (kitchen, office, ..).

930 **To identify stairs** we cluster connected components of the stair mesh extracted in Sec 4.2. For each
931 component we add an edge to the scene graph between the rooms/floors it connects.

932 **D.1 Identifying Building Floors**

933 We use the floor-classified polygons of the layout prototype and merge close levels with similar
934 heights. Specifically, we create a graph where the nodes represent floor-classified polygons and add
935 edges between polygons whose height differs by at most 50cm. Each connected component of the
936 graph defines a floor level. For each level, we determine its average *elevation* from its assigned floor
937 polygons.

938 **D.2 Creating a 2D Floorplan for each Level**

939 We construct each level’s 2D floorplan by computing the union of each level’s floor polygon(s) with
940 suitable ceiling polygons. Suitable ceiling polygons are identified by assigning each ceiling polygon
941 to the closest next-lower floor polygon that is at least 1m below the ceiling’s center. The level’s 2D
942 *floorplan* is then constructed from the union of the ceilings and the floors of the level.

943 We further identify a level’s *walls* by selecting wall-classified polygons that (1) intersect the floor’s
944 2D *floorplan* in BEV and (2) vertically intersect the height interval of $[0, 2.5]$ m above the floor’s
945 *elevation*.

946 **D.3 Segmenting each Level into Rooms**

947 We partition each level’s 2D *floorplan* into *rooms* using the level’s *walls*. We do so by applying
948 HovSG [15]’s morphology-based room segmentation algorithm twice: first with a bottleneck width
949 of 2.5m and then with a bottleneck width of 1.5m. The two-stage application has the benefit that
950 cells (rooms) with a diameter between 1.5m and 2.5m can exist individually, yet larger cells with
951 bottlenecks below 2.5m are separated.

952 Note that Hov-SG is a system designed for robotic navigation, and it does not reconstruct an explicit
953 layout, neither in 2D nor in 3D. Originally, it obtains the 2D floorplan by simply thresholding the
954 point density of the target floor.

955 We then follow HovSG in constructing a 2D *scene graph* with the *rooms* as nodes and the bottlenecks
956 that split them as edges. We consider an edge a *door* if its bottleneck width is below 1.5m, and an
957 *opening* otherwise. Each node has a 2D *floorplan* consisting of a cell of the entire level’s floorplan.

958 **D.4 Scene Graph Classification and Pruning**

959 We follow HovSG in computing a single, CLIP-aligned feature per room. For this, we use
960 OpenSeg [37] to compute pixel-aligned vision-language model features. Then we follow the same
961 steps as in the mesh segmentation (Sec. 4.2) to project the features to our mesh vertices. The per-
962 room feature is computed from the average mesh vertex features per room. We then use the CLIP
963 embeddings to classify the rooms into ‘bathroom’, ‘bedroom’, ‘living room’, ‘garage’, ‘entrance’,
964 ‘kitchen’, ‘office’, ‘stairs’, ‘gym’, ‘classroom’, ‘spa/sauna’, ‘mirror’, ‘grass/bushes/trees’, ‘driveway’,
965 and ‘veranda/terrace/balcony’.

966 We then use this classification to remove leaf nodes of the scene graph belonging to one of the last five
967 classes: This serves as an additional safeguard against the inclusion of outdoor spaces. (Additional to
968 the segmentation performed in Sec. 4.2). Notably, this pruning step contributes to the performance
969 improvement in Tab. 4 of the full method compared to the version without room segmentation.

D.5 Stair Detection

We combine the *stair mesh* obtained in Sec. 4.2 with the floor segmentation from Sec. 4.4 to identify stairs and approximate them as simple 3D rectangles.

That is, we cluster the stair mesh (*i.e.* the sub-mesh of stair-classified vertices in Sec. 4.2) into connected components. Each component’s vertices are now projected onto the horizontal plane and approximated by an oriented bounding rectangle R . We now assign the shorter edges of R to rooms of the scene graph by (1) determining the heights of the shorter edge midpoints by interpolation on the 3D cluster vertices, (2) lifting the rectangle to 3D using the edge midpoint heights and (3) assigning it to the room with the shortest point-to-polygon distance (D_{pp} , defined in Sec. 4.3) to the edge midpoint. If this distance is greater than 50cm or both edges are assigned to the same room, we reject this cluster. Otherwise, we add a *stair* edge connecting the two rooms and store R as its geometry.

D.6 Handling Doors and Stairs during Floor Extrusion

The described room extrusion algorithm produces a closed shell for each room by extruding its 2D floorplan. To connect the rooms with doors and stairs, we introduce openings into these shells as follows:

To extrude doors, we approximate the room-splitting boundaries obtained by HovSG with oriented 2D bounding rectangles. From these bounding rectangles, we build a 3D doorframe of fixed height 2.10m consisting of four rectangular faces. During room extrusion, we ensure the doorframe remains empty by only adding wall triangles above the door for edge fragments inside the door’s 2D bounding rectangle.

To extrude stairs, we first - in 2D - subtract the stair geometry from all rooms it intersects to avoid extruding overlapping regions. Then we extrude the stair geometry analogously to a room, with the difference that we as a last step adjust the height of the four floor corners in the resulting mesh to match the different levels it connects. We do not add walls for shared boundaries between stairs and rooms. For visualization, we add stair steps at a fixed stair step height on top of the otherwise pitched but flat floor of the stairs.

E Baseline Implementations

We compare our method to two recent, end-to-end trained baseline methods. Neither of the baselines is designed to predict multi-floor layouts. Adapting them to multiple floors is non-trivial. Hence, we use the ground-truth Matterport3D (MP3D) [7] segmentation of houses into levels and regions (loosely corresponding to rooms). We evaluate the baselines both on individual rooms and levels, concatenating the output.

E.1 RoomFormer

RoomFormer [1] predicts a 2D layout based on point clouds. We hence create its input by sampling points from the surface of the mesh. The prediction is then lifted to 3D by assuming a planar floor/ceiling, whose height we determine based on the 5%-quantile of the distribution of the heights of the input points. Doors are assumed to extend from floor level to 2.10m above floor level. Windows are assumed to span 80% of the height of a wall (centered).

E.2 SceneScript

SceneScript [2] predicts a set of 3D walls, doors, and windows based on *semidense* point clouds. We sample semi-dense from the input mesh by sampling points from the surface of the mesh where the norm of the surface gradient falls into the top 5%-quantile of all surface gradient norms. We further observe that the output improves when additionally sampling a small fraction of random points from all surfaces. We therefore incorporate random points into the input.

SceneScript neither predicts ceilings nor floors. We therefore infer one single floor and ceiling polygon respectively from the 2D Birds-Eye View convex hull of the output. To determine floor and ceiling height, we use the highest and lowest wall rectangles respectively.