Vectorizing Building Blueprints [Supplementary Material]

Weilian Song, Mahsa Maleki Abyaneh, Mohammad Amin Shabani, and Yasutaka Furukawa

Simon Fraser University, BC, Canada

The supplementary provides more experimental results (Figures 1 and 2), the neural architecture specification (Figure 3), and the system details (Section 1), which are not the technical core of the paper but are included here for reproducibility.

1 System details

1.1 Instance segmentation

We use the U-Net [4] architecture, with implementation borrowed from [1]. The input to the network is a $256 \times 256 \times 1$ grayscale blueprint crop, and the output is a $256 \times 256 \times 8$ embedding matrix.

1.2 Type classification

We use the Resnet50 [2] network architecture from the torchvision python library. We modify the network to take in a matrix of $224 \times 224 \times 2$, and output probability distribution over 7 classes (ignoring the background class). We utilize a mix of ground-truth (GT) and predicted data for training, assigning the corresponding GT type to each predicted instance. To determine correspondence, we overlay the predicted instance mask and the GT type segmentation, and choose the most frequent pixel type within the overlap. One exception to this rule is with thin and small instances, which are sometimes incorrectly assigned as rooms. To address this, we assign room type only when 70% or more overlapped pixels are of the type "room", otherwise we assign the second most frequent type.

1.3 Frame detection

We again use the Resnet50 [2] network architecture, with $224 \times 224 \times 10$ as input and output dimension of 8. We also perform mixed GT and predicted data training, assigning the corresponding GT connectivity vector for each predicted door/window/open-portal. To determine correspondence, we overlay the predicted instance mask and the GT instance segmentation, and choose the GT instance with the largest overlap and of the same type. If there is no correspondence, we simply skip the predicted instance during training.

1.4 Frame correction

We utilize the following heuristics to compute approximate centroid locations for missing frames. Missing frames are detected when comparing an instance's actual and predicted connectivity vector. As an initial candidate, we consider the side of the instance that the frame is touching (as obtained from our four-side heuristics), and interpolate the edge half-way to obtain its mid-point. We perform the interpolation using Shapely Python library's interpolate function. Given that most frames are connected to a wall, we then consider all walls touching the corresponding side and mark the approximate centroid as the point on the walls closest to the mid-point. If no walls are touching, then the mid-point becomes the approximate centroid.

During generator input preprocessing, for boundary hiding we perform binary dilation and erosion with a 3×3 kernel. We use the Scipy implementation of these functions. However, erosion often removes thin and small structures. To circumvent this: when the erosion operation removes more than 50% of the instance, we use scikit-image's skeletonize function to shrink the instance instead.

When painting a square mask for a missing frame or for hiding an existing frame during training, we randomly perturb the frame centroid location by up to two pixels in each of the four cardinal directions. This augmentation step seeks to let the generator learn the proper location of frames, instead of simply taking the mid-point of the square mask.

The architecture specification for our generator and discriminator are given in Figure 3. They are very similar to HouseGAN++ [3], but with blueprint image as an extra input. In addition, each stage of message passing receives extra information, in contrast to HouseGAN++ where all information are inputs at the beginning of the network.

At test time during iterative refinement, we hold constant a border of 8 pixels around the square crop. This is to prevent our generator from distorting results near the border, as they are ambiguous without more context.

References

1. Unet: semantic segmentation with pytorch. https://github.com/milesial/Pytorch-UNet, accessed: 2021-04-16 1
2. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 770–778 (2016) 1
3. Nauata, N., Hosseini, S., Chang, K.H., Chu, H., Cheng, C.Y., Furukawa, Y.: House-gan++: Generative adversarial layout refinement networks. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) (2021) 2, 5
4. Ronneberger, O., Fischer, P., Brox, T.: U-net: Convolutional networks for biomedical image segmentation. In: Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. pp. 234–241 (2015) 1

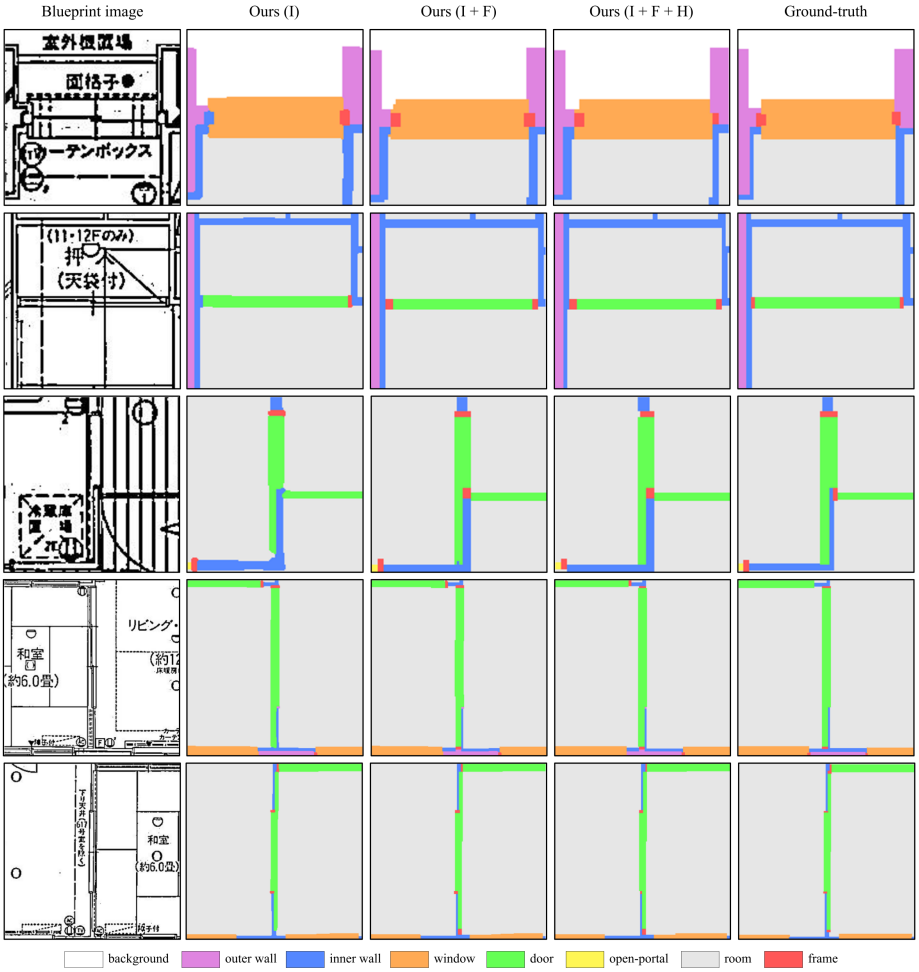

Fig. 1. Step-by-step visualization of our pipeline, focusing in around instances. For step names, "I" refers to the instance segmentation and type classification, "F" refers to the frame detection and the correction modules, and "H" refers to the heuristic simplification.

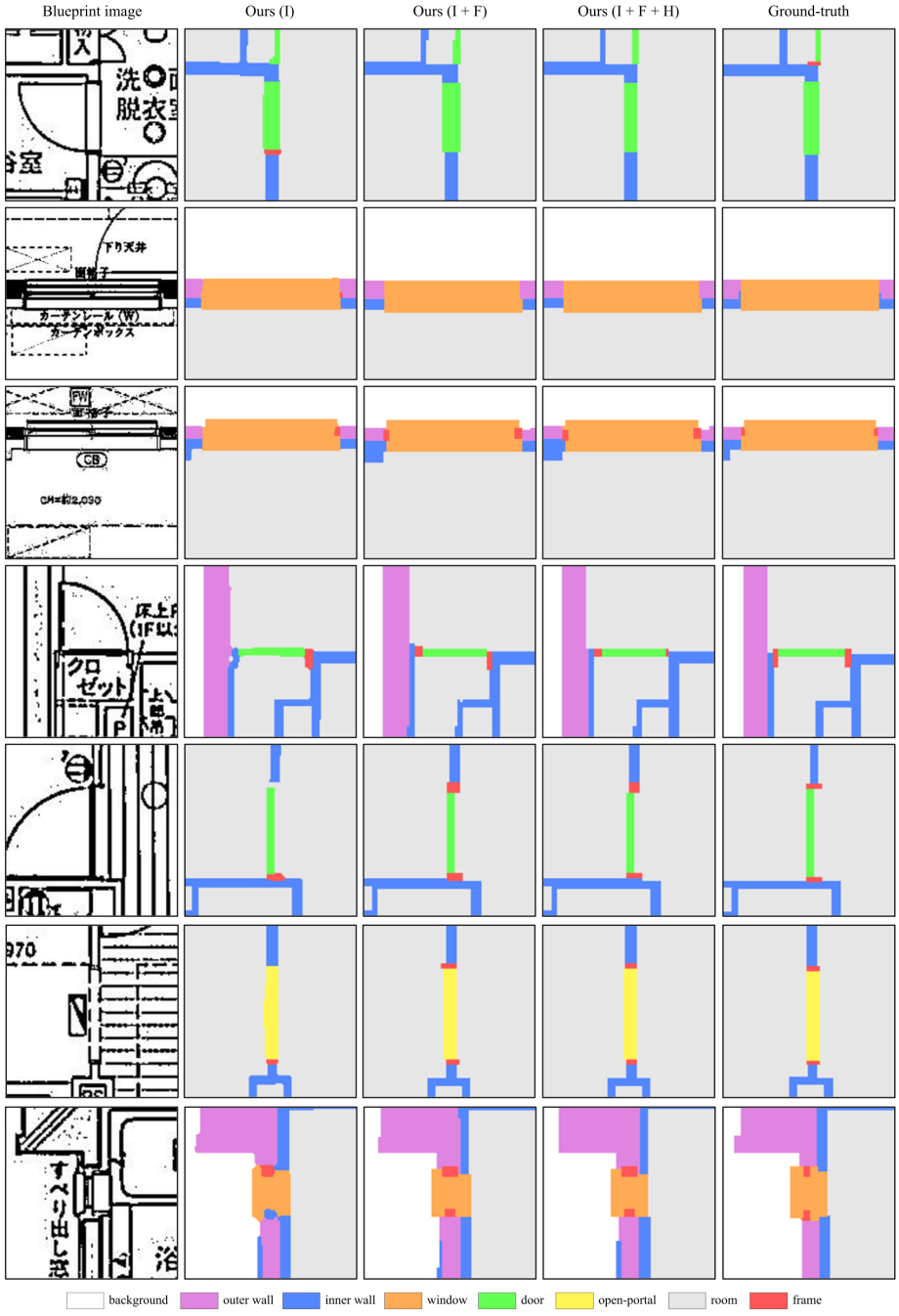

Fig. 2. Continued.

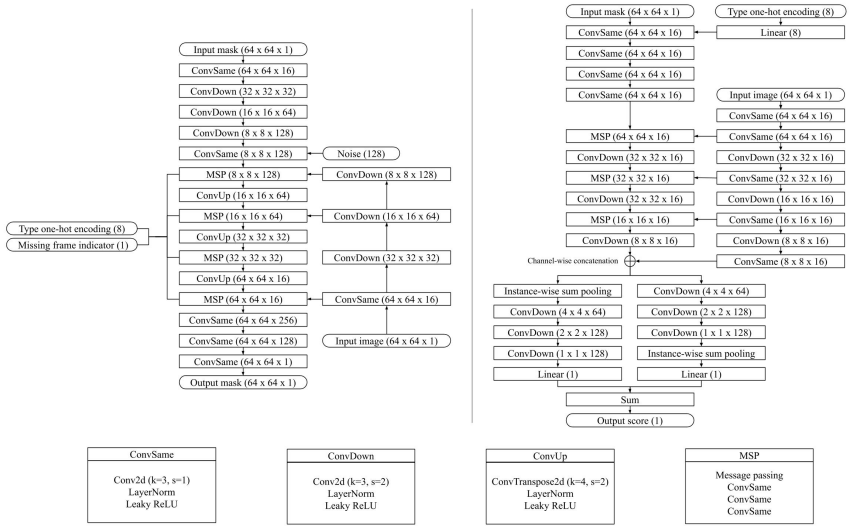

Fig. 3. Generator and discriminator architectures for our frame correction module. Square boxes indicate network layers, while rounded boxes indicate feature volumes. "k" stands for kernel size, "s" stands for stride, all convolutional layers use padding of 1, and layer output sizes are indicated in parentheses. For all vector inputs, each vector is copied to every pixel in the spatial dimension. Our architectures are very similar to HouseGAN++ [3], with two notable differences: we pass in a blueprint image as input, and provide extra information at each stage of message passing instead of at the beginning of the networks.