Worldsheet: Wrapping the World in a 3D Sheet for View Synthesis from a Single Image
(Supplementary Material)

A. Continuous and large viewpoint changes

Our approach allows synthesizing continuous novel views by smoothly moving to a new camera pose that is largely different from the input. We kindly request the readers to view the videos at `worldsheet.github.io` to better understand the performance of our method. In these videos, we compare our synthesized novel views (from a single image) to SynSin [7] on the RealEstate10K dataset with simulated large view-point changes (the first frame contains the input view and the rest of the frames are synthesized). From the videos, it can be seen that our model can generate novel views with much larger camera translation and rotation than in the training data, while SynSin often suffers from severe artifacts in these cases, likely because its refinement network does not generalize well to a sparser point cloud (resulting from large camera zoom-in or rotation).

We also show in these videos continuously synthesized novel views on high resolution (960×960) images over a wide range of scenes (the first frame is the input view), as described in our analysis in Sec. 4.3 in the main paper.

B. Ablation study: using depth supervision

In our experiments in the paper, we show that our model can be trained using only two views of a scene *without* 3D or depth supervision. In this section, we further analyze our approach by training it *with* depth supervision on the Matterport dataset, where the ground-truth depth can be obtained from the Habitat simulator.

In this analysis, we modify the differentiable mesh renderer to render RGB-D images from our mesh, and apply an L1 loss between the ground-truth and the rendered depth as additional supervision. We also compare with the performance of SynSin with depth supervision (reported in [7]). The results are shown in Table B.1. It can be seen that our model without depth supervision (the default setting; line 7) works almost equally as well as its counterpart using depth supervision (line 8) on the Matterport dataset and generalizes better to the Replica dataset. In addition, it outperforms SynSin under both supervision settings (lines 5-6).

C. Additional analyses on RealEstate10K

As described in Sec. 4.2 in the main paper, we follow the evaluation protocol of SynSin [7] on the RealEstate10K dataset. To enable comparison with StereoMag [8] that uses two input views on this dataset, in [7], the best metrics of two views were reported for single-view methods. At test time, for each target view, this involves making two separate predictions based on two different input views respectively, and then selecting the best metrics between the two predictions as the score for this target view. Note that this evaluation protocol is only applied to the RealEstate10K dataset (in Table 3 and 4 in the main paper) and is not applied to Matterport or Replica.

In this section, we further evaluate by taking the average metrics over all predictions to measure how well the model does on average from a single input view and to be consistent with our evaluation on Matterport and Replica in Table 1 and 2 in the main paper. Apart from metrics over the entire image, we would also like to analyze how well each model does on rendering regions seen in the input view vs. invisible regions (where things must be imagined). However, we cannot compute the exact visibility map as there are no ground-truth geometry annotations in the RealEstate10K dataset. To get an approximation, we evaluate on the central $\frac{W_{im}}{2} \times \frac{H_{im}}{2} = 128 \times 128$ crop of the target image (**Center**, which is nearly always visible) and the rest of the image (**Peripheral**, containing most of the invisible regions).

The results of these analyses are shown in Table C.1. It can be seen that our method achieves the highest performance on both the center regions (which are mostly visible) and the peripheral regions, outperforms the previous single-view based approaches by a large margin.

Our model uses a simple ResNet-50 backbone with an output stride of 8 pixels to extract image features (see Sec. 3.1 in the main paper), while SynSin [7] adopts a U-Net backbone that has a higher output feature resolution same as the input image (*i.e.* output feature stride is 1 pixel). To further study the impact of different backbone architectures, we train a variant of SynSin by replacing its U-Net backbone with the same ResNet-50 backbone pretrained on ImageNet (and upsampling its output feature map to stride 1 with a deconvolution layer) to be consistent with our model, shown in line 4 in Table C.1. Comparing it with line 3 or 6, it can be seen that this variant of SynSin with ResNet-50 backbone performs worse than the default SynSin architecture, or our model. This suggests that SynSin requires a high-resolution feature output to build a per-pixel point-cloud for view synthesis, while our model is able to work with a lower resolution (a larger stride) in the backbone.

#	Method	Matterport [1] PSNR ↑ Both	PSNR ↑ InVis	PSNR ↑ Vis	SSIM ↑ Both	SSIM ↑ InVis	SSIM ↑ Vis	Perc Sim ↓ Both	Perc Sim ↓ InVis	Perc Sim ↓ Vis	Replica [4] PSNR ↑	SSIM ↑	Perc Sim ↓
1	Im2Im [9]	15.87	16.20	15.97	0.53	0.60	0.48	2.99	0.58	2.05	17.42	0.66	2.29
2	Tatarchenko *et al*. [5]	14.79	14.83	15.05	0.57	0.62	0.53	3.73	0.74	2.50	14.36	0.68	3.36
3	Vox [3] w/ UNet	18.52	17.85	19.05	0.57	0.57	0.57	2.98	0.77	1.96	18.69	0.71	2.68
4	Vox [3] w/ ResNet	20.62	19.64	21.22	0.70	0.69	0.68	1.97	0.47	1.19	19.77	0.75	2.24
5	SynSin [7]	20.91	19.80	21.62	0.71	0.71	0.70	1.68	0.43	0.99	21.94	0.81	1.55
6	SynSin (w/ depth sup.) [7]	21.59	20.32	22.46	0.72	0.71	0.71	1.60	0.43	0.92	22.54	0.80	1.55
7	ours	**24.67**	**22.90**	**26.00**	**0.82**	**0.77**	**0.82**	**1.05**	**0.35**	**0.54**	**23.51**	**0.85**	**1.32**
8	ours (w/ depth sup.)	24.75	22.85	26.18	0.82	0.77	0.82	1.06	0.36	0.54	22.78	0.84	1.51

Table B.1: Analyses of our model and SynSin using explicit depth supervision during training (line 8 and 6) on the Matterport dataset. Our model without depth (the default setting; line 7) performs almost equally as well as its counter part with depth supervision (line 8) on Matterport and generalizes better to Replica, outperforming both variants of SynSin (line 5 and 6). See Sec. B for details.

#	Method	RealEstate10K [8] (averaged metrics over all predictions) PSNR ↑ Both	PSNR ↑ Peripheral	PSNR ↑ Center	SSIM ↑ Both	SSIM ↑ Peripheral	SSIM ↑ Center	Perc Sim ↓ Both	Perc Sim ↓ Peripheral	Perc Sim ↓ Center
1	Im2Im [9]	15.56	15.65	15.80	0.51	0.53	0.44	2.59	1.93	0.65
2	Tatarchenko *et al*. [5]	11.11	11.32	10.68	0.32	0.35	0.24	3.96	2.96	1.13
3	SynSin [7]	20.47	20.35	21.65	0.68	0.68	0.68	1.49	1.16	0.29
4	SynSin w/ R-50 backbone†	19.37	19.33	20.18	0.65	0.65	0.64	1.64	1.26	0.34
5	Single-View MPI [6]	21.17	20.90	23.03	0.70	0.70	0.71	1.59	1.27	0.31
6	ours (33×33 mesh)	23.11	22.90	24.83	**0.75**	0.74	**0.76**	1.17	0.94	**0.21**
7	ours (65×65 mesh)	**23.41**	**23.20**	**25.17**	**0.75**	**0.75**	**0.76**	**1.14**	**0.92**	**0.21**

Table C.1: Performance of our and previous approaches on the RealEstate10K dataset with metrics averaged over all predictions, which is consistent with our evaluation on Matterport and Replica in Sec. 4.1 in the main paper. We evaluate on the entire 256×256 image (**both**), the **center** 128×128 crop (nearly always visible), and the remaining **peripheral** regions (containing most of the invisible regions). See Sec. C for details. (†: We also evaluate a variant of SynSin by replacing its U-Net backbone with the same ResNet-50 backbone pretrained on ImageNet as used in our model.)

D. Details on differentiable texture sampler

Our differentiable texture sampler (Sec. 3.2 in the main paper) splats image pixels onto the texture map through each face. This splatting procedure involves three main steps: forward-mapping, normalization, and hole filling.

Suppose the flow $(u, v) = f(i, j)$ maps image coordinates (i, j) to UV coordinates (u, v) on the texture map, which can be obtained through mesh rasterization (here we drop the face index k for simplicity). To implement $\hat{T} = \text{splat}(I_{in}, f)$ (Eqn. 5 in the main paper), we first forward-map the image pixels to the texture map, using bilinear assignment when the mapped texture coordinates (u, v) do not fall on integers, as forward_map below.

```
function T = forward_map(I, f)
  T = all_zeros(W_uv, H_uv)
  for i in 1:W_im
    for j in 1:H_im
      u, v = f(i, j)
      uf, uc = floor(u), ceil(u)
      vf, vc = floor(v), ceil(v)
      T[uf,vf] += I[i,j] * (uc - u) * (vc - v)
      T[uc,vf] += I[i,j] * (u - uf) * (vc - v)
      T[uf,vc] += I[i,j] * (uc - u) * (v - vf)
      T[uc,vc] += I[i,j] * (u - uf) * (v - vf)
    end
  end
  return T
```

In the implementation above, gradients can be taken over f through the bilinear weights.

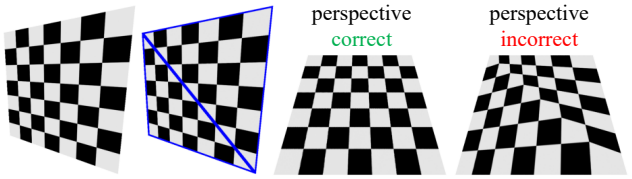
However, forward mapping alone will lead to incorrect pixel intensity (*e.g.* imagine down-scaling an image to half its width and height by forward-mapping – each pixel in the low-resolution image will receive assignment from 4 pixels and become $4\times$ brighter). Hence, a second normalization step is applied:

$$\hat{T}_{sum} = \text{forward\_map}(I_{in}, f) \quad \text{(D.1)}$$

$$\hat{W}_{sum} = \text{forward\_map}(I_{one}, f) \quad \text{(D.2)}$$

$$\hat{T}_{norm} = \hat{T}_{sum} / \max\left(\hat{W}_{sum}, 10^{-4}\right) \quad \text{(D.3)}$$

where I_{one} is an $W_{im} \times H_{im}$ image with all ones as its pixel



(a) input view (b) 2×2 mesh (c) novel view via our texture sampler (d) novel view via vertex *uv* on screen

Figure D.1: Our texture sampler is perspective-correct.

intensity. $\hat{T}_{norm}$ contains the normalized splatting result. A threshold 10^{-4} is applied to avoid division by zero (which could happen due to holes described below).

Filling holes with Gaussian filtering. It is well known that the bilinear forward mapping above often leads to holes in the output (*e.g.* imagine up-scaling an image to a much larger size – there will be gaps in the output image as some pixels will not receive assignments). To minimize hole occurrence, in our UV texture map we assign the UV coordinates of each mesh vertex with an equally-spaced $W_m \times H_m$ lattice grid, and use the same image size as the texture map size ($W_{uv} \times H_{uv} = W_{im} \times H_{im}$). This ensures that most texels on the texture map receive assignments in forward mapping, so that holes rarely occur in $\hat{T}_{norm}$. However, to address corner cases, we further apply a Gaussian filter to fill the holes in $\hat{T}_{norm}$ (where $\hat{W}_{sum}$ is zero as no assignment is received from forward-mapping):

$$\hat{M} = \mathrm{I}\left[\hat{W}_{sum} > 0\right] \tag{D.4}$$

$$\hat{T}_g = \left(F_g * \hat{T}_{norm}\right) / \left(F_g * \hat{M}\right) \tag{D.5}$$

$$\hat{T} = \hat{M} \cdot \hat{T}_{norm} + \left(\left(1 - \hat{M}\right) \cdot \hat{T}_g\right) \tag{D.6}$$

where F_g is a discrete 2D Gaussian kernel for image filtering (we use kernel size 7 and standard deviation 2 for F_g in our implementation). Here $\hat{M}$ is a binary mask indicating which pixels have received assignments in forward mapping (*i.e.* 1 means valid and 0 means holes), $\hat{T}_g$ is the Gaussian-blurred version of $\hat{T}_{norm}$ (where the division ensures the correct pixel intensity; otherwise it will be darker due to holes in $\hat{T}_{norm}$) and is used to fill only the holes in $\hat{T}_{norm}$. We use $\hat{T}$ in Eqn. D.6 as the final splatting output.

Perspective correctness. A main purpose of our differentiable texture sampler is to build perspective-correct novel views during texture reconstruction. We note that the alternative solution of directly using the input image as a texture map by putting vertex uv texture coordinates in the input screen space for mesh rendering breaks perspective correctness, as shown in Figure D.1 (d). For perspective-correct novel views, one needs to invert the texture-map-to-image perspective transform when building UV texture maps from the image, which we implement in our texture sampler shown in Figure D.1 (c).

E. Details on multi-layered Worldsheets

In our proposed Worldsheet model, we build a scene mesh by warping a planar sheet onto the scene. This model is capable of handling moderate occlusion and generating plausible novel views by deforming the mesh along object boundaries and refining the predicted novel view with an inpainting network. However, we also acknowledge that artifacts can sometimes occur in occluded regions or object boundaries when the disparity is very large between the input view and the novel view. This is partly because our current approach of deforming a mesh onto the scene does not capture all the fine-grained geometric details (such as the flower boundary as shown in the last two failure cases in the supplemental videos. We believe there is room for improvement in this direction (*e.g.* via adaptive resolution), which we are interested in exploring in future work.

In Sec. 3.5 in the main paper, we propose a simple extension with multi-layered Worldsheets. The main purpose of this extension is to separate objects or scene structures at different depth levels into different mesh layers, instead of placing them all on a single sheet. In this extension, the 3D mesh geometry of each layer provides the geometric support for the scene components, while the transparency channel in the RGBA texture maps allows segmentation between different components. For example, to represent a sofa object, a mesh layer can be wrapped onto a larger 3D surface region covering the sofa surface, with its texture map containing the sofa texture over the object region while being transparent on the surrounding regions.

Specifically, we predict and warp a total of L mesh sheets (*i.e.* L layers) onto the scene for view synthesis. For each layer $l = 1, \cdots, L$, we predict its grid offset $\left(\Delta\hat{x}^{(l)}_{w,h}, \Delta\hat{y}^{(l)}_{w,h}\right)$ and its depth $z^{(l)}_{w,h}$ from the convolutional feature map $\{q_{w,h}\}$ similar to Sec. 3.1, and also predict a pixel-wise transparency map $\alpha^{(l)}$ of size $H_{im} \times W_{im}$ in the screen space of the input view as follows.

$$\Delta\hat{x}^{(l)}_{w,h} = \frac{\tanh\left(W^{(l)}_1 q_{w,h} + b^{(l)}_1\right)}{W_m - 1} \tag{E.1}$$

$$\Delta\hat{y}^{(l)}_{w,h} = \frac{\tanh\left(W^{(l)}_2 q_{w,h} + b^{(l)}_2\right)}{H_m - 1} \tag{E.2}$$

$$z^{(l)}_{w,h} = g\left(W^{(l)}_3 q_{w,h} + b^{(l)}_3\right) \tag{E.3}$$

$$\left\{\alpha^{(l)}_{i,j}\right\} = \sigma\left(\mathrm{deconv}\left(\{q_{w,h}\}; W^{(l)}_4, b^{(l)}_4\right)\right) \tag{E.4}$$

Here $\alpha^{(l)}_{i,j}$ is the scalar alpha (*i.e.* transparency) value at image pixel (i, j), deconv is a deconvolution (*i.e.* transposed convolution) layer over the feature map with an output size equal to the image size, and $\sigma(\cdot)$ is the sigmoid function to transform the alpha values to the range between 0 and 1.

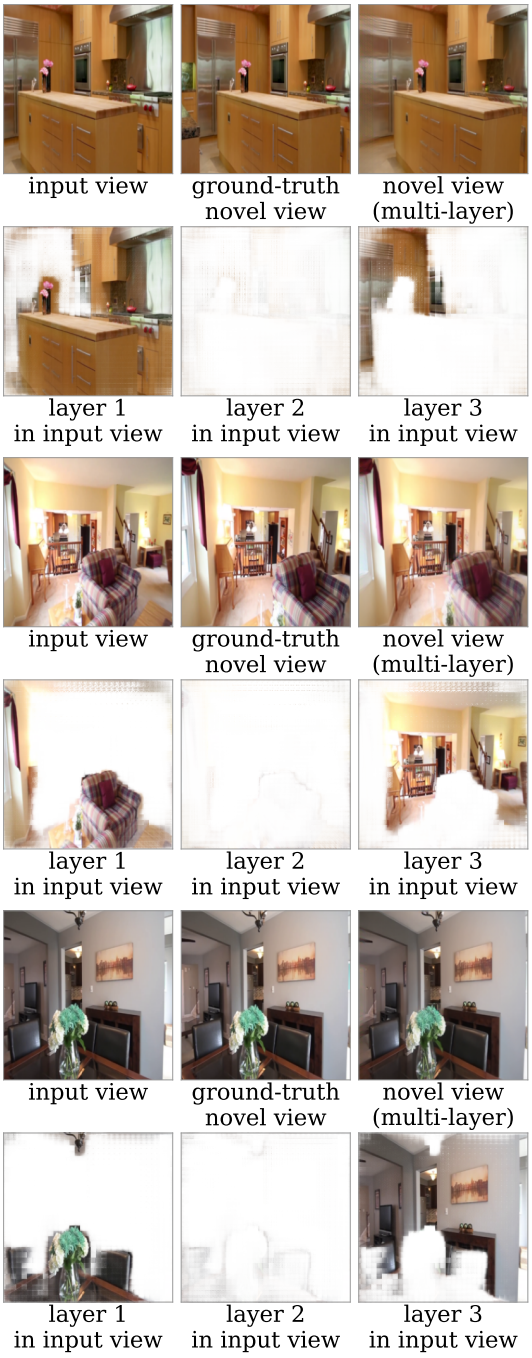


Figure E.1: View synthesis examples on RealEstate10K from our extension on multi-layered Worldsheets (see Sec. E for details). The top rows show the predicted novel views, while the bottom rows visualize each mesh layer $l = 1, ..., L$ (here $L = 3$) along with its RGBA texture map in the input view (white is transparent). Through end-to-end training with only 2D rendering losses, the model learns to place scene structures at different depth levels onto different mesh layers, and separate foreground objects (*e.g.* kitchen counter, sofa, or table) from their background.

For each layer $l = 1, \cdots, L$, we construct a corresponding 3D mesh sheet $M^{(l)}$ following the procedure in Sec. 3.1 and also build its UV texture map $\hat{T}^{(l)}$ consisting of RGBA channels by concatenating the predicted transparency values $\alpha_{i,j}^{(l)}$ with the input image and splatting them onto the mesh texture space using our differentiable texture sampler in Sec. 3.2. Finally, we render all the mesh faces from all L layers $M^{(1)}, \cdots, M^{(L)}$ in the novel view along with their RGBA UV texture maps $\hat{T}^{(1)}, \cdots, \hat{T}^{(L)}$ through alpha compositing. The whole model can be trained end-to-end under the same supervision using only 2D rendering losses.

We use a total of $L = 3$ layers in our analyses. Figure E.1 visualizes this extension on multi-layered Worldsheets. It can be seen that through end-to-end training, the model learns to place scene structures at different depth levels onto different mesh layers and separate foreground objects (*e.g.* kitchen counter, sofa, or table) from their background. We qualitatively find that it better handles occlusions and parallax effect under large viewpoint changes, as shown in Sec. 4.4 in the main paper.

F. Hyper-parameters in our model

In our nonlinear function $g(\cdot)$ to scale the network prediction into depth values (in Eqn. 3 in the main paper), we use different output scales based on the depth range in each dataset. On Matterport and Replica, we use

$$g(\psi) = 1/(0.75 \cdot \sigma(\psi) + 0.01) - 1. \tag{F.1}$$

On RealEstate10K (which has larger depth range), we double the output depth scale and use

$$g(\psi) = 2/(0.75 \cdot \sigma(\psi) + 0.01) - 2. \tag{F.2}$$

However, we find that the performance of our model is quite insensitive to the hyper-parameters in $g(\cdot)$.

In our differentiable texture sampler and the mesh renderer, we mostly follow the hyper-parameters in PyTorch3D [2]. We use $K = 10$ faces per pixel and 1e-8 blur radius in mesh rasterization, 1e-4 sigma and 1e-4 gamma in softmax RGB blending, and background color filled with the mean RGB intensity on each dataset. On Matterport and Replica, the input views have 90-degree field-of-view. On RealEstate10K, we multiply the actual camera intrinsic matrix of each frame into its camera extrinsic R and T matrices, so that we can still use the same intrinsics and 90-degree field-of-view in the renderer. On high resolution images in the wild (Sec. 4.3 in the main paper), we assume 45-degree field-of-view.

We choose our mesh size W_m and H_m based on the image size. In our experiments on Matterport and Replica (Sec. 4.1 in the main paper), we use 256×256 input image resolution following SynSin [7], and use pixel stride 8 on the lattice grid sheet (from which our mesh is built),

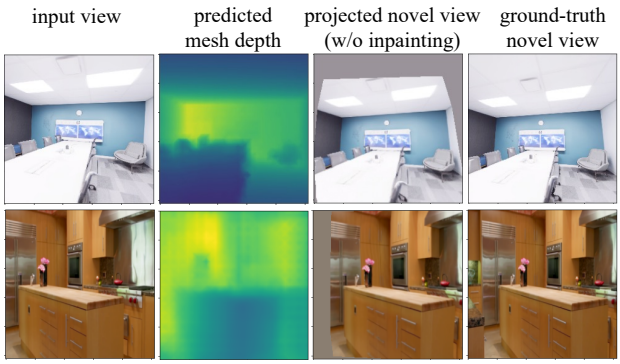


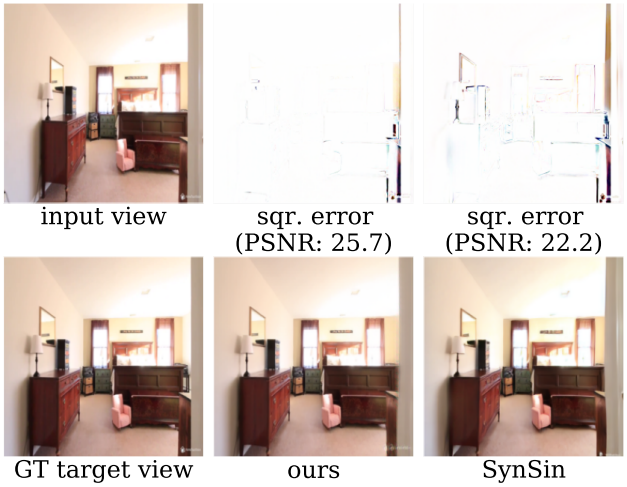
Figure G.1: Predicted depth maps from our scene mesh on Replica (upper) and RealEstate10K (lower).

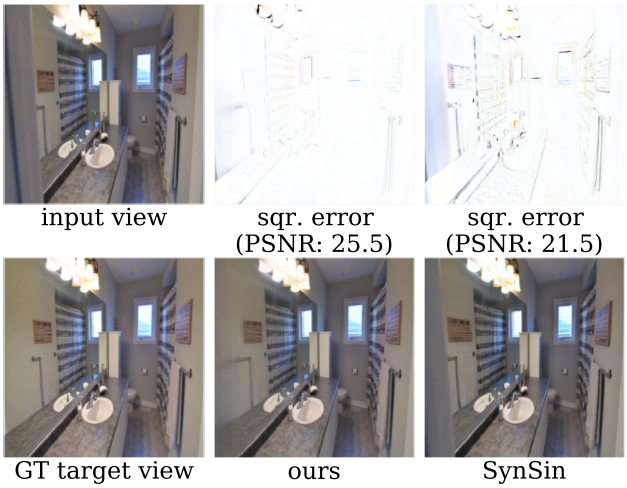
resulting in $W_m = H_m = 1 + 256/8 = 33$. On the RealEstate10K dataset (Sec. 4.2 in the main paper), we additionally experiment with pixel stride 4 on the grid sheet, giving $W_m = H_m = 1 + 256/4 = 65$. In our analysis on high resolution images in the wild (resized to have the image long side equal to 960 and padded to 960×960 square size for ease of rendering in PyTorch3D; Sec. 4.3 in the main paper), we use $W_m \times H_m = 129 \times 129$ mesh on the actual image regions (not including the padding regions).

G. More visualized examples

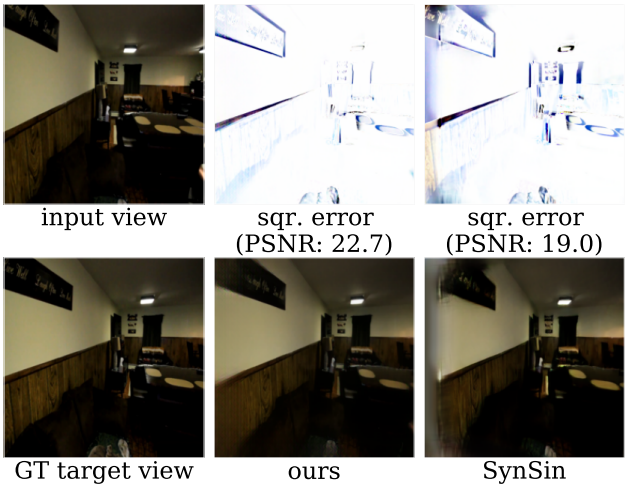
Figure G.1 shows the depth maps from our scene mesh, where most of the scene structure is captured, giving coherent novel view projections.

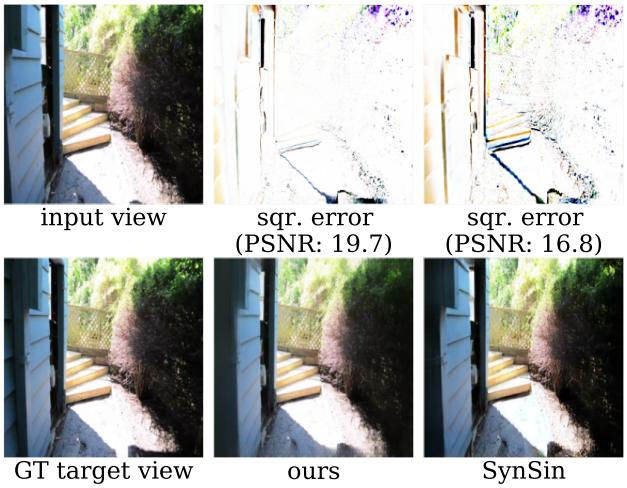
Figure G.2 shows additional visualization and error map comparisons between our approach and SynSin on the RealEstate10K dataset (similar to Figure 7 in the main paper), where our method paints things in the novel view at more precise locations with lower error and higher PSNR.


input view
sqr. error (PSNR: 25.7)
sqr. error (PSNR: 22.2)
GT target view
ours
SynSin


input view
sqr. error (PSNR: 25.5)
sqr. error (PSNR: 21.5)
GT target view
ours
SynSin


input view
sqr. error (PSNR: 24.5)
sqr. error (PSNR: 22.8)
GT target view
ours
SynSin


input view
sqr. error (PSNR: 22.7)
sqr. error (PSNR: 19.0)
GT target view
ours
SynSin


input view
sqr. error (PSNR: 19.7)
sqr. error (PSNR: 16.8)
GT target view
ours
SynSin

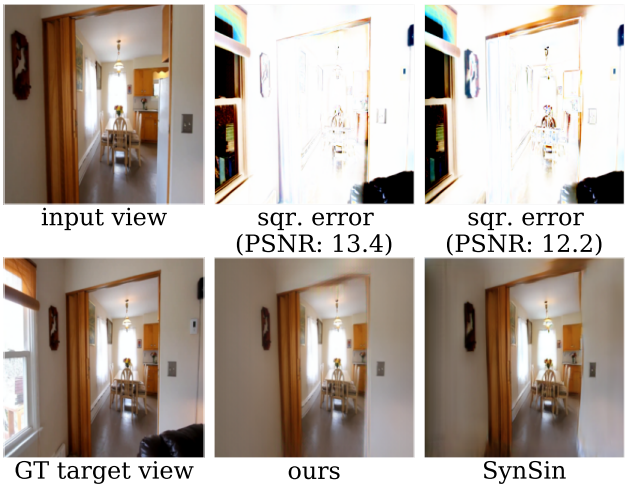

input view
sqr. error (PSNR: 13.4)
sqr. error (PSNR: 12.2)
GT target view
ours
SynSin

Figure G.2: Additional synthesized novel views (with squared error maps of the target view) from our method and SynSin [7] on the RealEstate10K dataset (darker is higher error). Our method paints things in the novel view at more precise locations, resulting in lower error and higher PSNR.

References

[1] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niebner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3d: Learning from rgb-d data in indoor environments. In *2017 International Conference on 3D Vision (3DV)*. IEEE, 2017, License for Matterport dataset available at http://kaldir.vc.in.tum.de/matterport/MP_TOS.pdf. 2

[2] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv preprint arXiv:2007.08501*, 2020. 4

[3] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhofer. Deepvoxels: Learning persistent 3d feature embeddings. In *CVPR*, 2019. 2

[4] Julian Straub, Thomas Whelan, Lingni Ma, Yufan Chen, Erik Wijmans, Simon Green, Jakob J Engel, Raul Mur-Artal, Carl Ren, Shobhit Verma, et al. The replica dataset: A digital replica of indoor spaces. *arXiv preprint arXiv:1906.05797*, 2019. 2

[5] Maxim Tatarchenko, Alexey Dosovitskiy, and Thomas Brox. Multi-view 3d models from single images with a convolutional network. In *ECCV*. Springer, 2016. 2

[6] Richard Tucker and Noah Snavely. Single-view view synthesis with multiplane images. In *CVPR*, 2020. 2

[7] Olivia Wiles, Georgia Gkioxari, Richard Szeliski, and Justin Johnson. Synsin: End-to-end view synthesis from a single image. In *CVPR*, pages 7467–7477, 2020. 1, 2, 4, 7

[8] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification: learning view synthesis using multiplane images. *ACM Transactions on Graphics (TOG)*, 2018. 1, 2

[9] Tinghui Zhou, Shubham Tulsiani, Weilun Sun, Jitendra Malik, and Alexei A Efros. View synthesis by appearance flow. In *ECCV*. Springer, 2016. 2