

Supplementary Materials: Learning Geometry Consistent Neural Radiance Fields from Sparse and Unposed Views

Anonymous Authors

A IMPLEMENTATION DETAILS

In this section, we illustrate more details about the training, evaluation, and implementation.

A.1 NeRF Architecture

We follow the standard NeRF [3] to set the network architecture. We also modify the MLP network with some modifications. For GC-NeRF, there is only a coarse MLP to calculate the volume density and RGB color of 3D coordinates. Moreover, we adopt the softplus activation before getting the volume density of the sampled points for improving stability. Each layer of the MLP network has 256 hidden units, except the last layer has three hidden units to get the RGB color. Notably, for a fair comparison, we remove the ground-truth depth map for all methods.

A.2 Matching Points

GC-NeRF sets itself apart from previous methods by requiring only a minimal percentage (e.g., 1% - 2%) of reliable matching point relationships between image pairs, as opposed to the 10% or higher demanded by prior works [6]. To achieve this, we leverage a pre-trained LoFTR [5] model for extracting matching relationships. The choice of LoFTR is motivated by its ability to generate high-quality matching correspondences even in challenging scenarios like images with weak textures or motion blur. This, in turn, paves the way for the practical deployment of NeRF from sparse and unposed views. In the presented approach, we proactively compute the matching point relationships for image pair I_S and I_T , utilizing the sufficient matching points and the corresponding confidence provided by LoFTR. Notably, GC-NeRF filters out the matching associations with a confidence below 0.95, to ensure the robustness of the pseudo-label. For example, on the LLFF dataset, we observe an average of 3000-4000 sets of matching points per image pair.

A.3 Implementation Details

Here, we describe the input settings and training details in our experiments.

Image Size. For the fair comparison, we follow the previous work [1, 3, 6] to resize the input images to 540*960, 378*504, and 378*504 of Tanks and Temples, LLFF, and NeRF Real 360 datasets, respectively. Specifically, all baselines and GC-NeRF have the same input settings in all experiments.

Loss Weighting. The training of GC-NeRF is split into two stages. First, we only use the intra-view image photometric supervision (Eq. (5) in the manuscript) about 2K iterations to assist camera pose parameters representing discrepancies between each other. Then Eq. (5) - Eq. (9) supervise together to establish multi-view geometry consistency for optimizing camera pose and NeRF parameters jointly about 8K iterations. In the first stage, the intra-view image photometric loss has a weight of 1 consistently. We set the weights λ_1 , λ_2 , and λ_3 to 10^{-4} , 10^{-3} , and 10^{-3} , respectively. Second,

we freeze camera pose parameters and finetune the NeRF model with four supervisions until rendering high-quality novel view images about 10K iterations. In the second stage, the intra-view image photometric loss also has a weight of 1 consistently. We set the weights λ_1 , λ_2 , and λ_3 to 10^{-3} , 10^{-4} , and 10^{-4} , respectively.

A.4 Evaluation Metrics

We elaborate on the details of the equations for metrics below, which are stemmed from previous research [1, 3, 6].

For Novel View Synthesis, we select Peak Signal-to-Noise Ratio (PSNR) [1], Structural Similarity Index Measure (SSIM) [7], and Learned Perceptual Image Patch Similarity (LPIPS) [8] standard evaluation metrics. The details are as follows:

$$\begin{aligned} \text{PSNR} &= 10 \cdot \log_{10} \left(\frac{255^2}{\text{MSE}} \right), \\ \text{MSE} &= \frac{1}{HW} \sum_{i=1}^{HW} (\hat{C}_i - C_i), \end{aligned} \quad (1)$$

where H and W denote the height and width of image size. We calculate the mean value of three channels. And \cdot depicts the multiplication operation.

$$\text{SSIM} = \frac{1}{N} \sum_{i=1}^N \frac{(2\mu_S\mu_T + c_1)(2\sigma_{ST} + c_2)}{(\mu_S^2 + \mu_T^2 + c_1)(\sigma_S^2 + \sigma_T^2 + c_2)}, \quad (2)$$

where μ denotes the mean value. σ_{ST} depicts the covariance. σ_S and σ_T are the variance. c_1 and c_2 are 1×10^{-4} and 9×10^{-4} , respectively. Specifically, we set the size of the slide window as 11×11 . And N is the number of slide window.

$$\text{LPIPS} = \frac{1}{HW} \sum_{i=1}^{HW} \|w \cdot (\hat{C}_i - C_i)\|_2^2, \quad (3)$$

where w is the coefficient to scale the activations channel wise. The settings is as the same as [3, 8].

For Camera Pose Estimation, we employ Relative Pose Error (RPE) to calculate the relative pose errors between image pairs, which includes rotation and translation errors [2, 4]. The details are as follows:

$$\text{RPE} = ([\mathbf{R}_S \ \mathbf{t}_S]^{-1} [\mathbf{R}_T \ \mathbf{t}_T])^{-1} ([\hat{\mathbf{R}}_S \ \hat{\mathbf{t}}_S]^{-1} [\hat{\mathbf{R}}_T \ \hat{\mathbf{t}}_T]), \quad (4)$$

where S and T are the index of image pairs. Then we calculate the RPE_R and RPE_t according to RPE, respectively. Notably, rotation errors are in degree, and translation errors are multiplied by 100.

For Depth Estimation, we follow [1] to adopt the Mean Depth Absolute Error (MDAE) for comparing the rendered and the ground-truth depth values. The details are as follows:

$$\text{MDAE} = \frac{1}{HW} \sum_{i=1}^{HW} \|\hat{d}_i - d_i\|_2^2, \quad (5)$$

where \hat{d} is predicted depth of pixel i . d is the GT depth of pixel i .

B QUALITATIVE COMPARISONS

In this section, we exhibit the qualitative comparisons of GC-NeRF, SPARF, NoPeNeRF, and GT results, which include the input RGB images, the output RGB images, and the output depth map. The comparison is illustrated in Fig. 1. Notably, GC-NeRF produces the most impressive renderings with significantly reduced artifacts and improved surface clarity. The learned scene geometry is also notably sharper and more accurate, as reflected in the depth map renderings. This observation is further supported in Fig. 2 - 4, where we provide additional renderings generated by GC-NeRF, SPARF, and NoPeNeRF across three challenging real-world datasets.

C MORE VISUALIZATIONS OF NOVEL VIEW SYNTHESIS

In this section, we report more visualization of GC-NeRF, SPARF, NoPeNeRF, and GT results, which contain the rendered RGB images and the rendered depth map. The results of three real-world datasets are illustrated in Fig. 2, Fig. 3, and Fig. 4, respectively. The results depict that GC-NeRF outperforms all baselines consistently. It verifies the superiority of image- and region-level geometric consistencies for GC-NeRF.

D LIMITATION

We further discuss the limitation of GC-NeRF, which we will focus on in future work. Although GC-NeRF significantly assists the standard NeRF to achieve SOTA performance by learning the geometry consistent from sparse and unposed views. It also needs 4-5 hours to train NeRF and camera pose jointly on a single NVIDIA RTX 3090 GPU for one scene. However, the efficiency is insufficient to support real-time joint optimization of NeRF on consumer-grade devices. Inspired by prior techniques, we aim to improve GC-NeRF by leveraging existing acceleration strategies (e.g. baked) in the future. This will enable GC-NeRF to significantly reduce computational demands and improve efficiency.

E CODE SEGMENT

Below, we present the key code snippets of GC-NeRF to facilitate understanding the details of framework.

```

22 torch.div(sample_pixel_S, W,
23           rounding_mode='floor']], dim=-1) #(N, 2)
24
25 sample_pixel_grid_T = sample_pixel_grid_S.clone() #TtoS
26
27
28 # project the coordinates to the other image
29 pose_w2c_S_4X4= homogeneous(pose_w2c_S)
30 pose_w2c_T_4X4= homogeneous(pose_w2c_T)
31
32 ret_S = self.net.render_image_at_specific_pose_and_rays(
33     self.opt, data_dict, pose_w2c_S,
34     intr_S, H, W, pixels=sample_pixel_grid_S.float(),
35     mode='train', iter=iteration)
36
37 ret_T = self.net.render_image_at_specific_pose_and_rays(
38     self.opt, data_dict, pose_w2c_T,
39     intr_T, H, W, pixels=sample_pixel_grid_T.float(),
40     mode='train', iter=iteration)
41
42 depth_S = ret_S['depth'].squeeze()
43 T_StoT = pose_w2c_T_4X4 @ \
44     pose_inverse_4x4(pose_w2c_S_4X4)
45 sample_pixel_grid_StoT, sample_depth_StoT = \
46     batch_project_to_other_img(
47     sample_pixel_grid_S.float(), depth_S, intr_S,
48     intr_T, T_StoT, return_depth=True) # (N, 2)
49
50 depth_T = ret_T['depth'].squeeze()
51 T_TtoS = pose_w2c_S_4X4 @ \
52     pose_inverse_4x4(pose_w2c_T_4X4)
53 sample_pixel_grid_TtoS, sample_depth_TtoS = \
54     batch_project_to_other_img(
55     sample_pixel_grid_T.float(), depth_T, intr_T,
56     intr_S, T_TtoS, return_depth=True) # (N, 2)
57
58
59 # get out-of-bounds mask
60 valid_pixels_StoT = sample_pixel_grid_StoT[:,0].ge(0) & \
61     sample_pixel_grid_StoT[:, 1].ge(0) & \
62     sample_pixel_grid_StoT[:, 0].le(W-1) & \
63     sample_pixel_grid_StoT[:, 1].le(H-1) & \
64     sample_depth_StoT.ge(data_dict.depth_range[T_idx][0])
65
66 sample_pixel_grid_StoT = \
67     sample_pixel_grid_StoT[valid_pixels_StoT]
68 sample_pixel_grid_S = \
69     sample_pixel_grid_S[valid_pixels_StoT]
70 sample_depth_StoT = sample_depth_StoT[valid_pixels_StoT]
71
72 valid_pixels_TtoS = sample_pixel_grid_TtoS[:,0].ge(0) & \
73     sample_pixel_grid_TtoS[:, 1].ge(0) & \
74     sample_pixel_grid_TtoS[:, 0].le(W-1) & \
75     sample_pixel_grid_TtoS[:, 1].le(H-1) & \
76     sample_depth_TtoS.ge(data_dict.depth_range[S_idx][0])
77
78 sample_pixel_grid_TtoS = \
79     sample_pixel_grid_TtoS[valid_pixels_TtoS]
80 sample_pixel_grid_T = \
81     sample_pixel_grid_T[valid_pixels_TtoS]
82 sample_depth_TtoS = sample_depth_TtoS[valid_pixels_TtoS]
83
84
85 # visibility mask
86 with torch.no_grad():
87     ret_max_depth_at_sampled_T = self.net. \
88         render_up_to_maxdepth_at_specific_pose_and_rays\
89         (self.opt, data_dict, pose_w2c_T, intr_T, H, W,
90          depth_max=sample_depth_StoT,

```

```

233 91         pixels=sample_pixel_grid_StoT.float(),
234 92         mode='train', iter=iteration)
235 93         visibility_weight_T = \
236 94         ret_max_depth_at_sampled_T['all_cumulated'] \
237 95         .squeeze(0).unsqueeze(-1) # (N, 1)
238 96         assert visibility_weight_T.le(1.).all()
239 97
240 98         valid_mask_StoT = visibility_weight_T.ge(0.2).reshape(-1)
241 99
242 100        sample_pixel_grid_S = \
243 101        sample_pixel_grid_S[valid_mask_StoT]
244 102        sample_pixel_grid_StoT = \
245 103        sample_pixel_grid_StoT[valid_mask_StoT]
246 104        sample_depth_StoT = sample_depth_StoT[valid_mask_StoT]
247 105
248 106        with torch.no_grad():
249 107            ret_max_depth_at_sampled_S = self.net. \
250 108            render_up_to_maxdepth_at_specific_pose_and_rays\
251 109            (self.opt, data_dict, pose_w2c_S, intr_S, H, W,
252 110             depth_max=sample_depth_TtoS,
253 111             pixels=sample_pixel_grid_TtoS.float(),
254 112             mode='train', iter=iteration)
255 113            visibility_weight_S = \
256 114            ret_max_depth_at_sampled_S['all_cumulated'] \
257 115            .squeeze(0).unsqueeze(-1) # (N, 1)
258 116            assert visibility_weight_S.le(1.).all()
259 117
260 118            valid_mask_TtoS = visibility_weight_S.ge(0.2).reshape(-1)
261 119
262 120            sample_pixel_grid_T = \
263 121            sample_pixel_grid_T[valid_mask_TtoS]
264 122            sample_pixel_grid_TtoS = \
265 123            sample_pixel_grid_TtoS[valid_mask_TtoS]
266 124            sample_depth_TtoS = sample_depth_TtoS[valid_mask_TtoS]
267 125
268 126            ret_StoT = self.net. \
269 127            render_image_at_specific_pose_and_rays(
270 128                self.opt, data_dict, pose_w2c_T, intr_T, H, W,
271 129                pixels=sample_pixel_grid_StoT.float(),
272 130                mode='train', iter=iteration)
273 131            ret_TtoS = self.net. \
274 132            render_image_at_specific_pose_and_rays(
275 133                self.opt, data_dict, pose_w2c_S, intr_S, H, W,
276 134                pixels=sample_pixel_grid_TtoS.float(),
277 135                mode='train', iter=iteration)
278 136
279 137            render_rgb_StoT = ret_StoT['rgb']
280 138
281 139            render_depth_StoT = ret_StoT['depth'].squeeze()
282 140            depth_ratio_StoT = render_depth_StoT / sample_depth_StoT
283 141
284 142            depth_ratio_StoT_mask = \
285 143            depth_ratio_StoT.ge(opt.depth_beta) & \
286 144            depth_ratio_StoT.le(1 / opt.depth_beta)
287 145            img_S = data_dict['image'][S_idx] \
288 146            .reshape(1, 3, -1).permute(0,2,1)
289 147
290 148            sample_pixel_grid_S_long = sample_pixel_grid_S.long()
291 149            sample_pixel_idx_S = \
292 150            sample_pixel_grid_S_long[:, 1]*W + \
293 151            sample_pixel_grid_S_long[:, 0]
294 152            rgb_S = img_S[:, sample_pixel_idx_S]
295 153
296 154            render_rgb_StoT = \
297 155            render_rgb_StoT[:, depth_ratio_StoT_mask]
298 156            rgb_S = rgb_S[:, depth_ratio_StoT_mask]
299 157
300 158            loss_aps_StoT = loss_func_aps(render_rgb_StoT, rgb_S)
301 159
302 160            render_rgb_TtoS = ret_TtoS['rgb']
303 161
304 162            render_depth_TtoS = ret_TtoS['depth'].squeeze()
305 163            depth_ratio_TtoS = render_depth_TtoS / sample_depth_TtoS
306 164            depth_ratio_TtoS_mask = \
307 165            depth_ratio_TtoS.ge(opt.depth_beta) & \
308 166            depth_ratio_TtoS.le(1 / opt.depth_beta)
309 167
310 168            img_T = data_dict['image'][T_idx] \
311 169            .reshape(1, 3, -1).permute(0,2,1)
312 170
313 171            sample_pixel_grid_T_long = sample_pixel_grid_T.long()
314 172            sample_pixel_idx_T = \
315 173            sample_pixel_grid_T_long[:, 1] * W + \
316 174            sample_pixel_grid_T_long[:, 0]
317 175            rgb_T = img_T[:, sample_pixel_idx_T]
318 176
319 177            render_rgb_TtoS = \
320 178            render_rgb_TtoS[:, depth_ratio_TtoS_mask]
321 179            rgb_T = rgb_T[:, depth_ratio_TtoS_mask]
322 180
323 181            loss_aps_TtoS = loss_func_aps(render_rgb_TtoS, rgb_T)
324 182
325 183            loss_aps = (loss_aps_StoT + loss_aps_TtoS) / 2
326 184
327 185            # compute loss_mps
328 186            _, _, H, W = data_dict.image.shape
329 187            id_S, id_matching_view, corres_map_S_to_T_, \
330 188            conf_map_S_to_T_, variance_S_to_T_, \
331 189            mask_correct_corr = self.sample_valid_image_pair()
332 190
333 191            # for correspondence loss
334 192            corres_map_S_to_T = corres_map_S_to_T_.detach()
335 193            conf_map_S_to_T = conf_map_S_to_T_.detach()
336 194            mask_correct_corr = \
337 195            mask_correct_corr.detach().squeeze(-1) # (H, W)
338 196            corres_map_S_to_T_rounded = \
339 197            torch.round(corres_map_S_to_T).long() # (H, W, 2)
340 198            corres_map_S_to_T_rounded_flat = \
341 199            corres_map_S_to_T_rounded[:, :, 1] * W + \
342 200            corres_map_S_to_T_rounded[:, :, 0] # (H, W)
343 201
344 202            pixels_in_S = self.grid[mask_correct_corr] # [N, 2]
345 203            ray_in_S_int = self.grid_flat[mask_correct_corr] # [N]
346 204
347 205            pixels_in_T = corres_map_S_to_T[mask_correct_corr]
348 206            ray_in_T_int = \
349 207            corres_map_S_to_T_rounded_flat[mask_correct_corr]
350 208            conf_values = conf_map_S_to_T[mask_correct_corr]
351 209
352 210            if ray_in_S_int.shape[0] > self.opt.nerf.rand_rays:
353 211                random_values = torch.randperm(ray_in_S_int.shape[0],
354 212                device=self.device)[:self.opt.nerf.rand_rays]
355 213                ray_in_S_int = ray_in_S_int[random_values]
356 214                pixels_in_S = pixels_in_S[random_values]
357 215
358 216            pixels_in_T = pixels_in_T[random_values]
359 217            ray_in_T_int = ray_in_T_int[random_values]
360 218            conf_values = conf_values[random_values]
361 219
362 220            pose_w2c_S, pose_w2c_T, intr_S, intr_T = \
363 221            get_ST_pose_intr(data_dict, id_S, id_matching_view)
364 222            pose_w2c_S = homogeneous(pose_w2c_S)
365 223            pose_w2c_T = homogeneous(pose_w2c_T)
366 224
367 225            ret_S = self.net.render_image_at_specific_pose_and_rays(
368 226                self.opt, data_dict, pose_w2c_S[:3], intr_S,
369 227                H, W, pixels=pixels_in_S, mode='train',
370 228

```

```

349 229         iter=iteration)
350 230
351 231     render_depth_S = ret_S['depth'].squeeze()
352 232
353 233     T_StoT = pose_w2c_T @ pose_inverse_4x4(pose_w2c_S)
354 234     pixels_StoT = batch_project_to_other_img(
355 235         pixels_in_S.float(), render_depth_S,
356 236         intr_S, intr_T, T_StoT, return_depth=False)
357 237     pixels_StoT_long = torch.round(pixels_StoT).long()
358 238     valid_pixels = pixels_StoT_long[:, 0].ge(0.) & \
359 239         pixels_StoT_long[:, 1].ge(0.) & \
360 240         pixels_StoT_long[:, 0].le(W-1) & \
361 241         pixels_StoT_long[:, 1].le(H-1)
362 242     pixels_StoT_long = pixels_StoT_long[valid_pixels]
363 243     pixels_idx_StoT = pixels_StoT_long[:, 1] * W + \
364 244         pixels_StoT_long[:, 0]
365 245
366 246     pixels_in_S_long = torch.round(pixels_in_S).long()
367 247     pixels_in_S_long = pixels_in_S_long[valid_pixels]
368 248     pixels_idx_S = pixels_in_S_long[:, 1] * W + \
369 249         pixels_in_S_long[:, 0]
370 250
371 251     img_S = data_dict['image'][id_S] \
372 252         .reshape(1, 3, -1).permute(0,2,1)
373 253     img_T = data_dict['image'][id_matching_view] \
374 254         .reshape(1, 3, -1).permute(0,2,1)
375 255
376 256     rgb_S = img_S[:, pixels_idx_S]
377 257     rgb_StoT = img_T[:, pixels_idx_StoT]
378 258
379 259     conf_values = conf_values[valid_pixels]
380 260
381 261     loss_mps = loss_func_mps(rgb_S, rgb_StoT, conf_values)
382 262
383 263     # compute loss_sss
384 264     _, _, H, W = data_dict.image.shape
385 265     id_S, id_matching_view, corres_map_S_to_T_, \
386 266         conf_map_S_to_T_, variance_S_to_T_, \
387 267         mask_correct_corr = self.sample_valid_image_pair()
388 268
389 269     # for correspondence loss
390 270     corres_map_S_to_T = corres_map_S_to_T_.detach()
391 271     conf_map_S_to_T = conf_map_S_to_T_.detach()
392 272     mask_correct_corr = \
393 273         mask_correct_corr.detach().squeeze(-1) # (H, W)
394 274     corres_map_S_to_T_rounded = \
395 275         torch.round(corres_map_S_to_T).long() # (H, W, 2)
396 276     corres_map_S_to_T_rounded_flat = \
397 277         corres_map_S_to_T_rounded[:, :, 1] * W + \
398 278         corres_map_S_to_T_rounded[:, :, 0] # (H, W)
399 279
400 280     pixels_in_S = self.grid[mask_correct_corr] # [N, 2]
401 281     ray_in_S_int = self.grid_flat[mask_correct_corr] # [N]
402 282
403 283     pixels_in_T = corres_map_S_to_T[mask_correct_corr]
404 284     ray_in_T_int = \
405 285         corres_map_S_to_T_rounded_flat[mask_correct_corr]
406 286     conf_values = conf_map_S_to_T[mask_correct_corr]
407 287
408 288     if ray_in_S_int.shape[0] > self.opt.nerf.rand_rays:
409 289         random_values = torch.randperm(
410 290             ray_in_S_int.shape[0],
411 291             device=self.device)[:self.opt.nerf.rand_rays]
412 292     ray_in_S_int = ray_in_S_int[random_values]
413 293     pixels_in_S = pixels_in_S[random_values]
414 294
415 295     pixels_in_T = pixels_in_T[random_values]
416 296     ray_in_T_int = ray_in_T_int[random_values]
417 297
418 298     conf_values = conf_values[random_values]
419 299
420 300     pose_w2c_S, pose_w2c_T, intr_S, intr_T = \
421 301         get_ST_pose_intr(data_dict, id_S, id_matching_view)
422 302     pose_w2c_S = homogeneous(pose_w2c_S)
423 303     pose_w2c_T = homogeneous(pose_w2c_T)
424 304
425 305     ret_S = self.net.render_image_at_specific_pose_and_rays(
426 306         self.opt, data_dict, pose_w2c_S[:3], intr_S, H, W,
427 307         pixels=pixels_in_S, mode='train', iter=iteration)
428 308     ret_T = self.net.render_image_at_specific_pose_and_rays(
429 309         self.opt, data_dict, pose_w2c_T[:3], intr_T, H, W,
430 310         pixels=pixels_in_T, mode='train', iter=iteration)
431 311
432 312     render_depth_S = ret_S['depth'].squeeze()
433 313     render_depth_T = ret_T['depth'].squeeze()
434 314
435 315     pose_c2w_S = pose_inverse_4x4(pose_w2c_S)
436 316     pose_c2w_T = pose_inverse_4x4(pose_w2c_T)
437 317
438 318     pts3d_in_w_from_S = batch_backproject_to_3d(
439 319         kpi=pixels_in_S, di=render_depth_S, Ki=intr_S,
440 320         T_itotj=pose_c2w_S)
441 321     pts3d_in_w_from_T = batch_backproject_to_3d(
442 322         kpi=pixels_in_T, di=render_depth_T, Ki=intr_T,
443 323         T_itotj=pose_c2w_T)
444 324
445 325     loss_sss = loss_func_sss(pts3d_in_w_from_S,
446 326         pts3d_in_w_from_T, conf_values)
447 327
448 328     # AdaRFF positional encoding
449 329     shape = input_position.shape
450 330     freq = 2**torch.arange(L, dtype=torch.float32,
451 331         device=input_position.device) * np.pi
452 332
453 333     input_position_freq =
454 334         input_position[...] * freq # [B,...,N,L]
455 335
456 336     pe_encoding = torch.stack([input_position_freq.sin(),
457 337         input_position_freq.cos()], dim=-2) # [B,...,N,2,L]
458 338     pe_encoding = pe_encoding.view(*shape[:-1],-1)#[B,..,2NL]
459 339
460 340     rff = rff_parm * input_position[...] * freq
461 341
462 342     rff_encoding = torch.stack([rff.sin(), rff.cos()],
463 343         dim=-2)
464 344     rff_encoding = rff_encoding.view(*shape[:-1], -1)
465 345
466 346     W1_APE = torch.matmul(rff_encoding, W1)
467 347     W2_PE = torch.matmul(pe_encoding, W2)
468 348
469 349     sigma = torch.sigmoid(W1_APE + W2_PE)
470 350
471 351     AdaRFF = torch.mul(sigma, rff_encoding) + \
472 352         torch.mul((1 - sigma), pe_encoding)
473 353
474 354
475 355
476 356
477 357
478 358
479 359
480 360
481 361
482 362
483 363
484 364
485 365
486 366
487 367
488 368
489 369
490 370
491 371
492 372
493 373
494 374
495 375
496 376
497 377
498 378
499 379
500 380
501 381
502 382
503 383
504 384
505 385
506 386
507 387
508 388
509 389
510 390
511 391
512 392
513 393
514 394
515 395
516 396
517 397
518 398
519 399
520 400
521 401
522 402
523 403
524 404
525 405
526 406

```

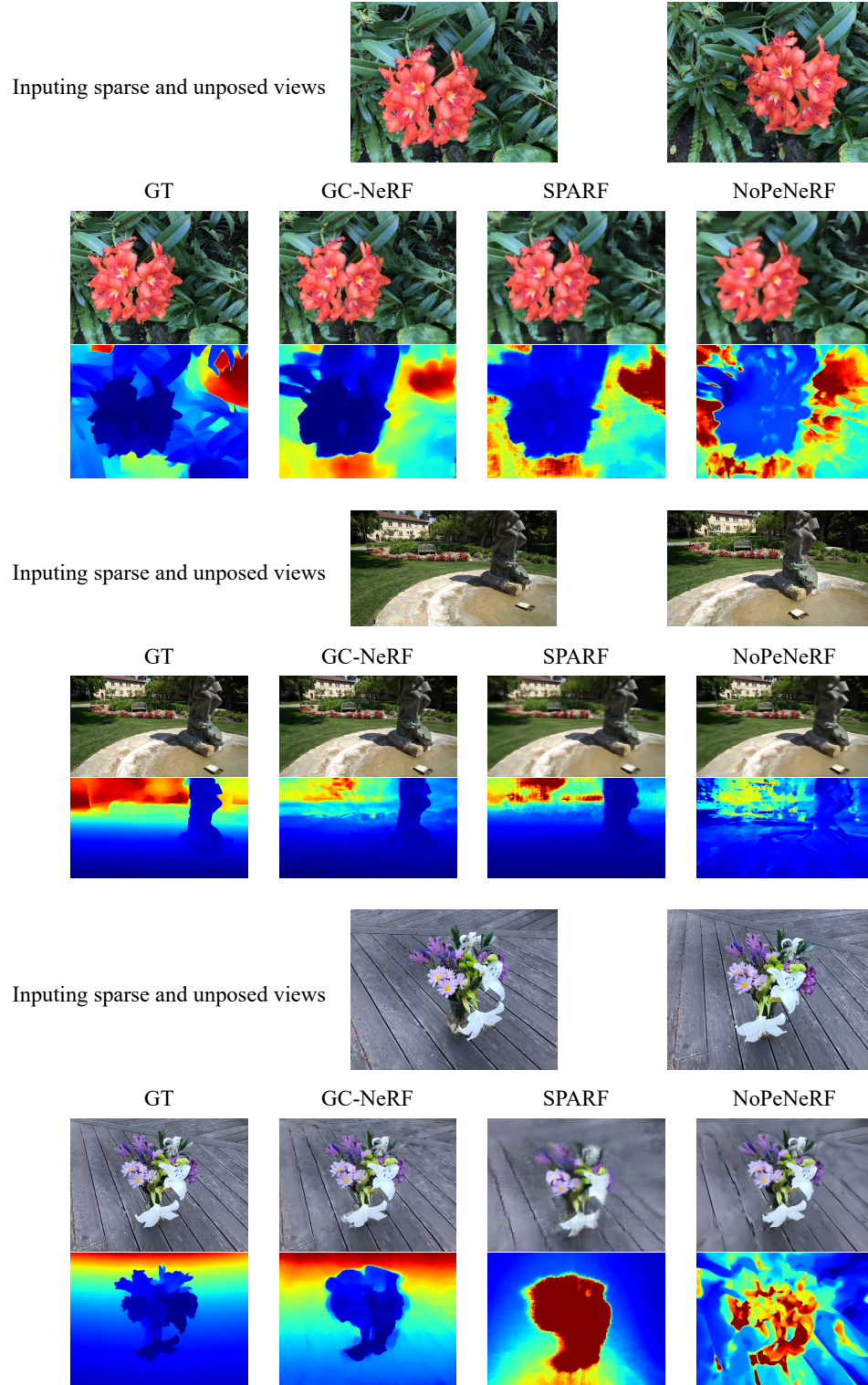



Figure 1: The qualitative comparisons of baselines and GC-NeRF with unknown pose images (2 views) about novel view rendering on Tanks and Temples, LLFF, NeRF Real 360 datasets, respectively.

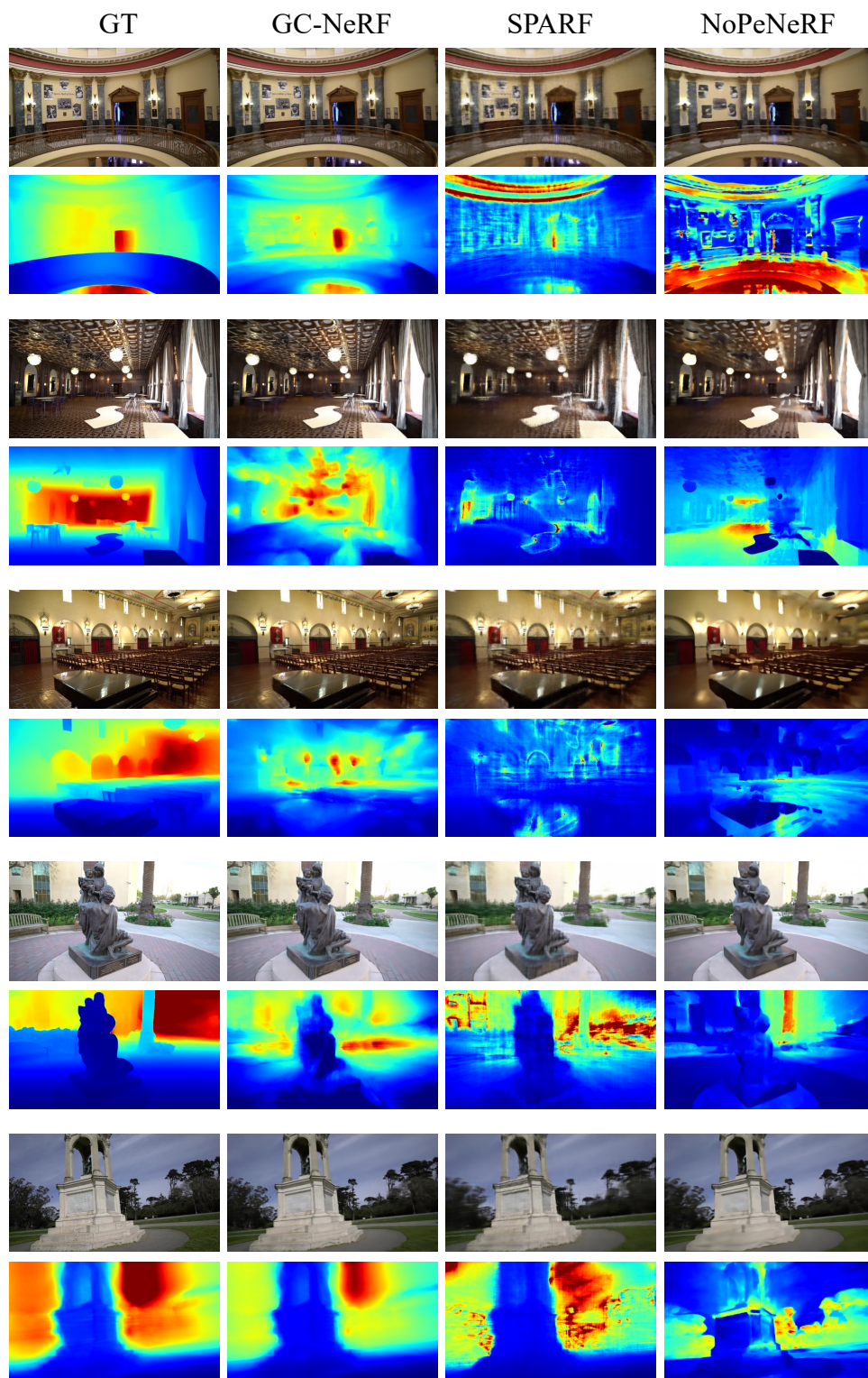


Figure 2: The visualization of baselines and GC-NeRF with unknown pose images (2 views) about novel view rendering on Tanks and Temples dataset.

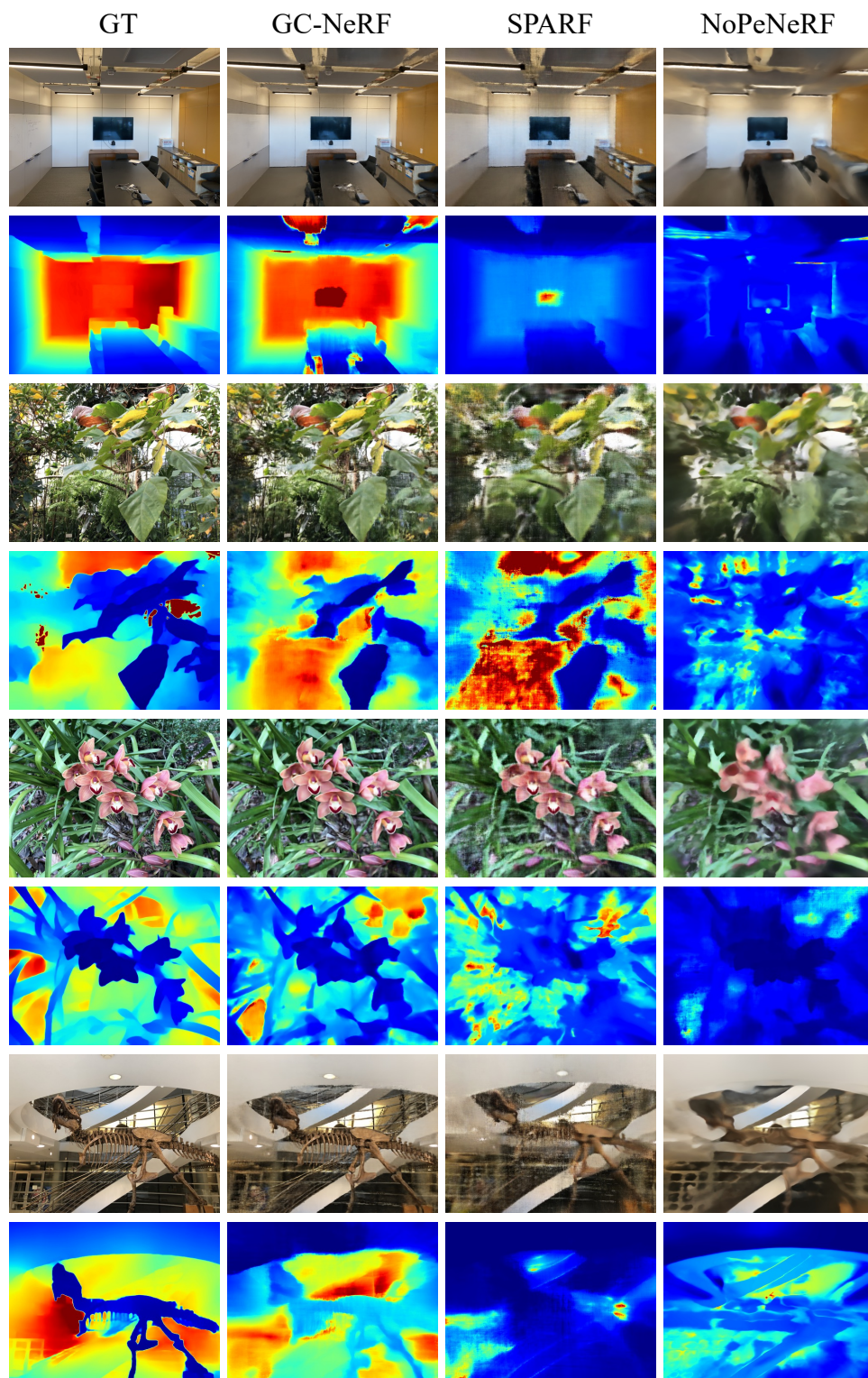


Figure 3: The visualization of baselines and GC-NeRF with unknown pose images (2 views) about novel view rendering on LLFF dataset.

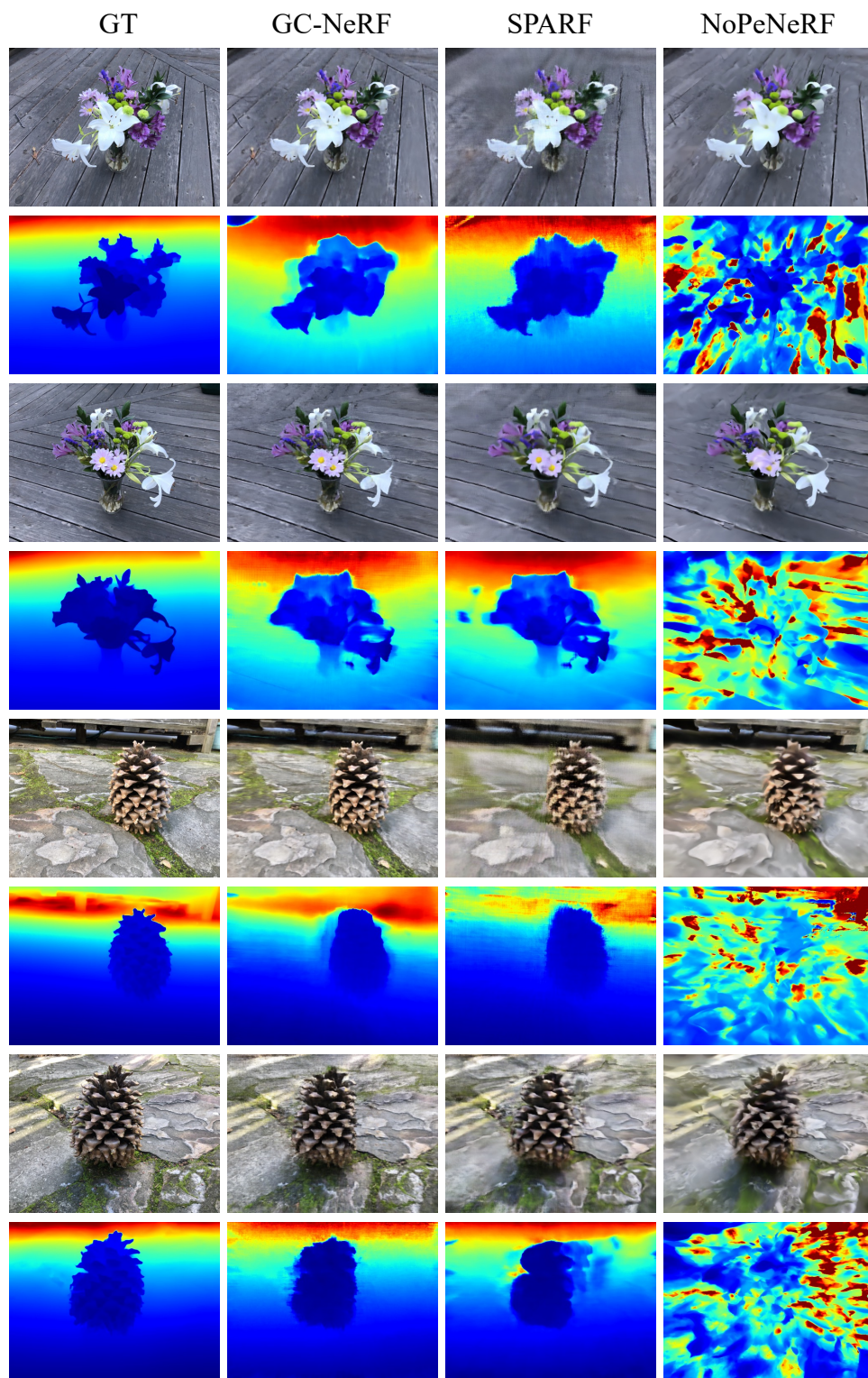


Figure 4: The visualization of baselines and GC-NeRF with unknown pose images (2 views) about novel view rendering on NeRF Real 360 dataset.

REFERENCES

- [1] Wenjing Bian, Zirui Wang, Kejie Li, and Jia-Wang Bian. 2023. NoPe-NeRF: Optimising Neural Radiance Field with No Pose Prior. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4160–4169.
- [2] Johannes Kopf, Xuejian Rong, and Jia-Bin Huang. 2021. Robust Consistent Video Depth Estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1611–1621.
- [3] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In *Proceedings of the 16th European Conference on Computer Vision*. 405–421.
- [4] Jürgen Sturm, Nikolas Engelhard, Felix Endres, Wolfram Burgard, and Daniel Cremers. 2012. A benchmark for the evaluation of RGB-D SLAM systems. In *Proceedings of the IEEE/RJS International Conference on Intelligent Robots and Systems*. 573–580.
- [5] Jiaming Sun, Zehong Shen, Yuang Wang, Hujun Bao, and Xiaowei Zhou. 2021. LoFTR: Detector-Free Local Feature Matching With Transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8922–8931.
- [6] Prune Truong, Marie-Julie Rakotosaona, Fabian Manhardt, and Federico Tombari. 2023. SPARF: Neural Radiance Fields from Sparse and Noisy Poses. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4190–4200.
- [7] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612.
- [8] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. 2018. The Unreasonable Effectiveness of Deep Features as a Perceptual Metric. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 586–595.