

## HW6 — 3D Deep Learning

# 1 Task 1: Estimating F and Epipoles

## 1.1 Coding: find\_fundamental\_matrix()

Code submitted to Canvas.

```
1 def find_fundamental_matrix(shape, pts1, pts2):
2     """
3     Computes Fundamental Matrix F that relates points in two images by the
4     :
5
6     [u' v' 1] F [u v 1]^T = 0
7     or
8     l = F [u v 1]^T -- the epipolar line for point [u v] in image 2
9     [u' v' 1] F = l' -- the epipolar line for point [u' v'] in image
10    1
11
12    Where (u,v) and (u',v') are the 2D image coordinates of the left and
13    the right images respectively.
14
15    Inputs:
16    - shape: Tuple containing shape of img1
17    - pts1: Numpy array of shape (N,2) giving image coordinates in img1
18    - pts2: Numpy array of shape (N,2) giving image coordinates in img2
19
20    Returns:
21    - F: Numpy array of shape (3,3) giving the fundamental matrix F
22    """
23
24    #This will give you an answer you can compare with
25    #Your answer should match closely once you've divided by the last
26    entry
27    FOpenCV, _ = cv2.findFundamentalMat(pts1, pts2, cv2.FM_8POINT)
28
29    F = np.eye(3)
30    #
31    #####
32
33    # TODO: Your code here
34    #
35    #
36    #####
```

```

30 # Normalize the points to increase accuracy
31 pts1_hom = homogenize(pts1)
32 pts2_hom = homogenize(pts2)
33
34 # Center and scale points for numerical stability
35 mean1 = np.mean(pts1, axis=0)
36 mean2 = np.mean(pts2, axis=0)
37 std1 = np.std(pts1)
38 std2 = np.std(pts2)
39
40 # Transformation matrices for normalization
41 T1 = np.array([
42     [1/std1, 0, -mean1[0]/std1],
43     [0, 1/std1, -mean1[1]/std1],
44     [0, 0, 1]
45 ])
46 T2 = np.array([
47     [1/std2, 0, -mean2[0]/std2],
48     [0, 1/std2, -mean2[1]/std2],
49     [0, 0, 1]
50 ])
51
52 # Normalize points
53 pts1_norm = (T1 @ pts1_hom.T).T
54 pts2_norm = (T2 @ pts2_hom.T).T
55
56 # Create matrix A for the linear equation system Ax = 0
57 A = np.zeros((len(pts1), 9))
58 for i in range(len(pts1)):
59     x1, y1, _ = pts1_norm[i]
60     x2, y2, _ = pts2_norm[i]
61     A[i] = [x2*x1, x2*y1, x2, y2*x1, y2*y1, y2, x1, y1, 1]
62
63 # Solve the homogeneous equation system using SVD
64 U, S, Vt = np.linalg.svd(A)
65 F = Vt[-1].reshape(3, 3)
66
67 # Enforce the rank constraint (rank 2)
68 U, S, Vt = np.linalg.svd(F)
69 S[2] = 0 # Set smallest singular value to 0
70 F = U @ np.diag(S) @ Vt
71
72 # Denormalize the fundamental matrix
73 F = T2.T @ F @ T1
74 print("F error: ", np.sum(F - FOpenCV))
75 #
76 #####
77
78                                     END OF YOUR CODE
79
80 #####
81
82 return F

```

## 1.2 Coding: compute\_epipoles()

Code submitted to Canvas.

```

1 def compute_epipoles(F):
2     """
3     Given a Fundamental Matrix F, return the epipoles represented in
4     homogeneous coordinates.
5
6     Check: e2@F and F@e1 should be close to [0,0,0]
7
8     Inputs:
9     - F: the fundamental matrix
10
11     Return:
12     - e1: the epipole for image 1 in homogeneous coordinates
13     - e2: the epipole for image 2 in homogeneous coordinates
14     """
15     #
16     #####
17
18     # TODO: Your code here
19     #
20     #
21     #####
22
23     # Compute the right epipole (e2): null space of F
24     U, S, Vt = np.linalg.svd(F)
25     e2 = Vt[-1] + 1e-10 # The last row of V^T, corresponding to the
26     smallest singular value
27
28     # Compute the left epipole (e1): null space of F^T
29     U, S, Vt = np.linalg.svd(F.T)
30     e1 = U[:, -1] + 1e-10 # The last column of U, corresponding to the
31     smallest singular value
32     #
33     #####
34
35     #
36     #
37     #
38     #
39     #
40     #
41     #
42     #
43     #
44     #
45     #
46     #
47     #
48     #
49     #
50     #
51     #
52     #
53     #
54     #
55     #
56     #
57     #
58     #
59     #
60     #
61     #
62     #
63     #
64     #
65     #
66     #
67     #
68     #
69     #
70     #
71     #
72     #
73     #
74     #
75     #
76     #
77     #
78     #
79     #
80     #
81     #
82     #
83     #
84     #
85     #
86     #
87     #
88     #
89     #
90     #
91     #
92     #
93     #
94     #
95     #
96     #
97     #
98     #
99     #
100    #
101    #
102    #
103    #
104    #
105    #
106    #
107    #
108    #
109    #
110    #
111    #
112    #
113    #
114    #
115    #
116    #
117    #
118    #
119    #
120    #
121    #
122    #
123    #
124    #
125    #
126    #
127    #
128    #
129    #
130    #
131    #
132    #
133    #
134    #
135    #
136    #
137    #
138    #
139    #
140    #
141    #
142    #
143    #
144    #
145    #
146    #
147    #
148    #
149    #
150    #
151    #
152    #
153    #
154    #
155    #
156    #
157    #
158    #
159    #
160    #
161    #
162    #
163    #
164    #
165    #
166    #
167    #
168    #
169    #
170    #
171    #
172    #
173    #
174    #
175    #
176    #
177    #
178    #
179    #
180    #
181    #
182    #
183    #
184    #
185    #
186    #
187    #
188    #
189    #
190    #
191    #
192    #
193    #
194    #
195    #
196    #
197    #
198    #
199    #
200    #
201    #
202    #
203    #
204    #
205    #
206    #
207    #
208    #
209    #
210    #
211    #
212    #
213    #
214    #
215    #
216    #
217    #
218    #
219    #
220    #
221    #
222    #
223    #
224    #
225    #
226    #
227    #
228    #
229    #
230    #
231    #
232    #
233    #
234    #
235    #
236    #
237    #
238    #
239    #
240    #
241    #
242    #
243    #
244    #
245    #
246    #
247    #
248    #
249    #
250    #
251    #
252    #
253    #
254    #
255    #
256    #
257    #
258    #
259    #
260    #
261    #
262    #
263    #
264    #
265    #
266    #
267    #
268    #
269    #
270    #
271    #
272    #
273    #
274    #
275    #
276    #
277    #
278    #
279    #
280    #
281    #
282    #
283    #
284    #
285    #
286    #
287    #
288    #
289    #
290    #
291    #
292    #
293    #
294    #
295    #
296    #
297    #
298    #
299    #
300    #
301    #
302    #
303    #
304    #
305    #
306    #
307    #
308    #
309    #
310    #
311    #
312    #
313    #
314    #
315    #
316    #
317    #
318    #
319    #
320    #
321    #
322    #
323    #
324    #
325    #
326    #
327    #
328    #
329    #
330    #
331    #
332    #
333    #
334    #
335    #
336    #
337    #
338    #
339    #
340    #
341    #
342    #
343    #
344    #
345    #
346    #
347    #
348    #
349    #
350    #
351    #
352    #
353    #
354    #
355    #
356    #
357    #
358    #
359    #
360    #
361    #
362    #
363    #
364    #
365    #
366    #
367    #
368    #
369    #
370    #
371    #
372    #
373    #
374    #
375    #
376    #
377    #
378    #
379    #
380    #
381    #
382    #
383    #
384    #
385    #
386    #
387    #
388    #
389    #
390    #
391    #
392    #
393    #
394    #
395    #
396    #
397    #
398    #
399    #
400    #
401    #
402    #
403    #
404    #
405    #
406    #
407    #
408    #
409    #
410    #
411    #
412    #
413    #
414    #
415    #
416    #
417    #
418    #
419    #
420    #
421    #
422    #
423    #
424    #
425    #
426    #
427    #
428    #
429    #
430    #
431    #
432    #
433    #
434    #
435    #
436    #
437    #
438    #
439    #
440    #
441    #
442    #
443    #
444    #
445    #
446    #
447    #
448    #
449    #
450    #
451    #
452    #
453    #
454    #
455    #
456    #
457    #
458    #
459    #
460    #
461    #
462    #
463    #
464    #
465    #
466    #
467    #
468    #
469    #
470    #
471    #
472    #
473    #
474    #
475    #
476    #
477    #
478    #
479    #
480    #
481    #
482    #
483    #
484    #
485    #
486    #
487    #
488    #
489    #
490    #
491    #
492    #
493    #
494    #
495    #
496    #
497    #
498    #
499    #
500    #
501    #
502    #
503    #
504    #
505    #
506    #
507    #
508    #
509    #
510    #
511    #
512    #
513    #
514    #
515    #
516    #
517    #
518    #
519    #
520    #
521    #
522    #
523    #
524    #
525    #
526    #
527    #
528    #
529    #
530    #
531    #
532    #
533    #
534    #
535    #
536    #
537    #
538    #
539    #
540    #
541    #
542    #
543    #
544    #
545    #
546    #
547    #
548    #
549    #
550    #
551    #
552    #
553    #
554    #
555    #
556    #
557    #
558    #
559    #
560    #
561    #
562    #
563    #
564    #
565    #
566    #
567    #
568    #
569    #
570    #
571    #
572    #
573    #
574    #
575    #
576    #
577    #
578    #
579    #
580    #
581    #
582    #
583    #
584    #
585    #
586    #
587    #
588    #
589    #
590    #
591    #
592    #
593    #
594    #
595    #
596    #
597    #
598    #
599    #
600    #
601    #
602    #
603    #
604    #
605    #
606    #
607    #
608    #
609    #
610    #
611    #
612    #
613    #
614    #
615    #
616    #
617    #
618    #
619    #
620    #
621    #
622    #
623    #
624    #
625    #
626    #
627    #
628    #
629    #
630    #
631    #
632    #
633    #
634    #
635    #
636    #
637    #
638    #
639    #
640    #
641    #
642    #
643    #
644    #
645    #
646    #
647    #
648    #
649    #
650    #
651    #
652    #
653    #
654    #
655    #
656    #
657    #
658    #
659    #
660    #
661    #
662    #
663    #
664    #
665    #
666    #
667    #
668    #
669    #
670    #
671    #

```

### 1.3 Show epipolar lines for temple, reallyInwards, and another dataset of your choice.

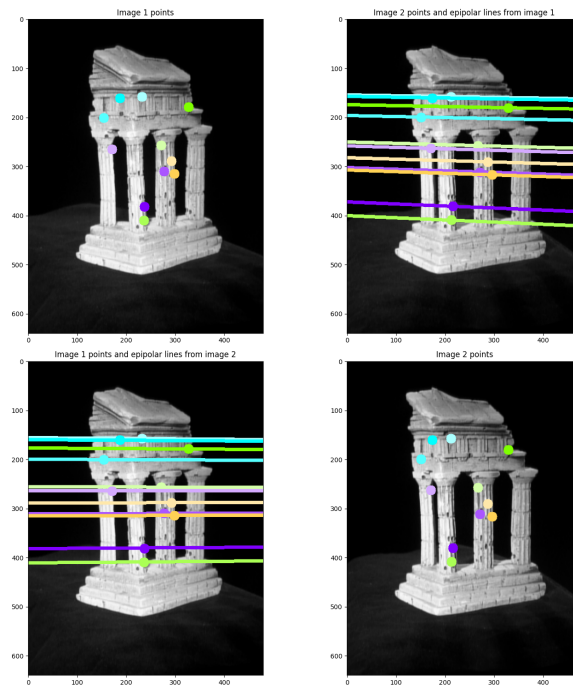


Figure 1: Temple

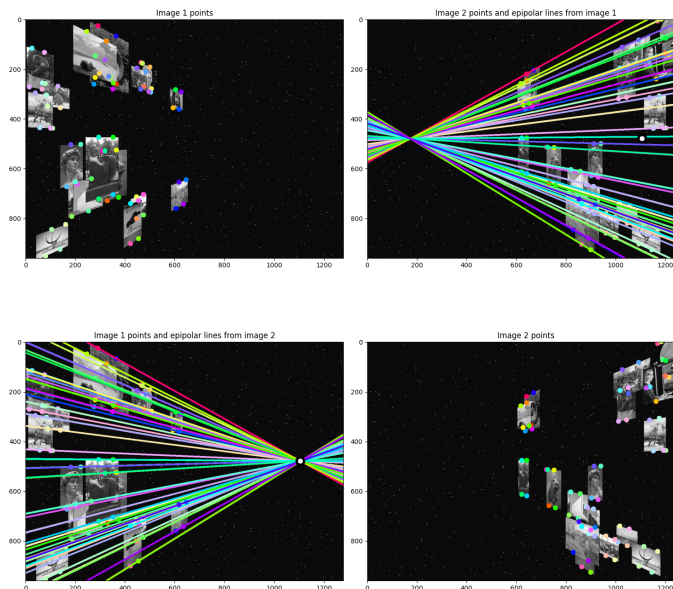


Figure 2: reallyInwards

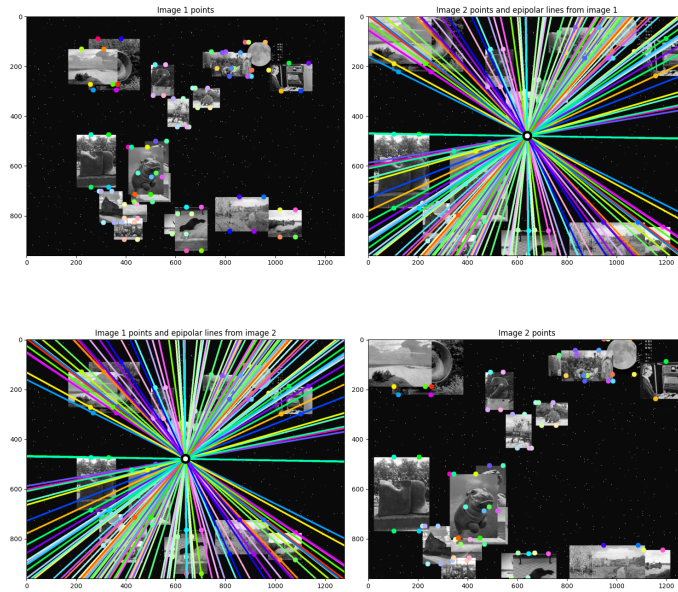


Figure 3: ztrans

## 1.4 Report the epipoles for reallyInwards and xtrans.

```

1 xtrans epipoles
2 [1.00000000e+00 9.99985793e-11 1.00000000e-10]
3 [-1.00000000e+00 1.00001421e-10 1.00000014e-10]
4 xtrans epipoles real coordinates
5 [1.00000000e+10 9.99985793e-01]
6 [-9.99999860e+09 1.00001407e+00]
7
8 reallyInwards epipoles
9 [9.17529952e-01 3.97665813e-01 8.29633180e-04]
10 [9.17529952e-01 3.97665813e-01 8.29633180e-04]
11 reallyInwards epipoles real coordinates
12 [1105.94654942 479.32727702]
13 [1105.94654942 479.32727702]

```

## 2 Task 2: NeRF

### 2.1 Coding: positional encoding

Code submitted to Canvas.

```

1 def positional_encoder(x, L_embed=6):
2     """
3     This function applies positional encoding to the input tensor.
4     Positional encoding is used in NeRF

```

```

4   to allow the model to learn high-frequency details more effectively. It
5   applies sinusoidal functions
6   at different frequencies to the input.
7
8   Parameters:
9   x (torch.Tensor): The input tensor to be positionally encoded.
10  L_embed (int): The number of frequency levels to use in the encoding.
11  Defaults to 6.
12
13  Returns:
14  torch.Tensor: The positionally encoded tensor.
15  """
16
17  # Initialize a list with the input tensor.
18  rets = [x]
19
20  # Loop over the number of frequency levels.
21  for i in range(L_embed):
22      #
23      #####
24
25      #                                TODO
26      #
27      #
28      #####
29
30      rets.append(torch.sin(2 ** i * x))
31      rets.append(torch.cos(2 ** i * x))
32      #
33      #####
34
35      #                                END OF YOUR CODE
36      #
37      #
38      #####
39
40  # Concatenate the original and encoded features along the last dimension
41  .
42  return torch.cat(rets, -1)

```

## 2.2 Coding: render()

Code submitted to Canvas.

```

1  def render(model, rays_o, rays_d, near, far, n_samples, rand=False):
2      """
3      Render a scene using a Neural Radiance Field (NeRF) model. This function
4      samples points along rays,
5      evaluates the NeRF model at these points, and applies volume rendering
6      techniques to produce an image.

```

```

6 Parameters:
7 model (torch.nn.Module): The NeRF model to be used for rendering.
8 rays_o (torch.Tensor): Origins of the rays.
9 rays_d (torch.Tensor): Directions of the rays.
10 near (float): Near bound for depth sampling along the rays.
11 far (float): Far bound for depth sampling along the rays.
12 n_samples (int): Number of samples to take along each ray.
13 rand (bool): If True, randomize sample depths. Default is False.
14
15 Returns:
16 tuple: A tuple containing the RGB map and depth map of the rendered
17       scene.
18
19 # Sample points along each ray, from 'near' to 'far'.
20 z = torch.linspace(near, far, n_samples).to(device)
21 if rand:
22     mids = 0.5 * (z[..., 1:] + z[..., :-1])
23     upper = torch.cat([mids, z[..., -1:]], -1)
24     lower = torch.cat([z[..., :1], mids], -1)
25     t_rand = torch.rand(z.shape).to(device)
26     z = lower + (upper - lower) * t_rand
27
28 #
29     #####
30
31     #
32     #
33     #
34     #####
35
36 # Compute 3D coordinates of the sampled points along the rays.
37 points = rays_o[..., None, :] + rays_d[..., None, :] * z[..., :, None]
38 #
39     #####
40
41 #
42     #
43     #
44     #####
45
46 # Flatten the points and apply positional encoding.
47 flat_points = torch.reshape(points, [-1, points.shape[-1]])
48 flat_points = positional_encoder(flat_points)
49
50 # Evaluate the model on the encoded points in chunks to manage memory
51 usage.
52 chunk = 1024 * 32
53 raw = torch.cat([model(flat_points[i:i + chunk]) for i in range(0,
54     flat_points.shape[0], chunk)], 0)
55 raw = torch.reshape(raw, list(points.shape[:-1]) + [4])
56
57 # Compute densities (sigmas) and RGB values from the model's output.

```

```

47 sigma = F.relu(raw[..., 3])
48 rgb = torch.sigmoid(raw[..., :3])
49
50 # Perform volume rendering to obtain the weights of each point.
51 one_e_10 = torch.tensor([1e10], dtype=rays_o.dtype).to(device)
52 dists = torch.cat((z[..., 1:] - z[..., :-1], one_e_10.expand(z[..., :1].
    shape)), dim=-1)
53 alpha = 1. - torch.exp(-sigma * dists)
54 weights = alpha * cumprod_exclusive(1. - alpha + 1e-10)
55
56 #
57     #####
58
59     #
60     #
61     #
62     #####
63
64 # Compute the weighted sum of RGB values along each ray to get the final
    pixel color.
65 rgb_map = torch.sum(rgb * weights[..., None], dim=-2)
66 # Compute the depth map as the weighted sum of sampled depths.
67 depth_map = torch.sum(weights * z, dim=-1)
68 #
69     #####
70
71 #
72     END OF YOUR CODE
73     #
74     #
75     #####
76
77 return rgb_map, depth_map

```

## 2.3 Coding: train()

Code submitted to Canvas.

```

1 mse2psnr = lambda x : -10. * torch.log(x) / torch.log(torch.Tensor([10.]))
    .to(device)
2
3 def train(model, optimizer, n_iters=3000):
4     """
5     Train the Neural Radiance Field (NeRF) model. This function performs
6     training over a specified number of iterations,
7     updating the model parameters to minimize the difference between
8     rendered and actual images.
9
10    Parameters:
11    model (torch.nn.Module): The NeRF model to be trained.
12    optimizer (torch.optim.Optimizer): The optimizer used for training the
13    model.
14    n_iters (int): The number of iterations to train the model. Default is
15    3000.

```



```

12  """
13
14  psnrs = []
15  iternums = []
16
17  plot_step = 500
18  n_samples = 64    # Number of samples along each ray.
19
20  for i in tqdm(range(n_iters)):
21      # Randomly select an image from the dataset and use it as the target
22      # for training.
23      images_idx = np.random.randint(images.shape[0])
24      target = images[images_idx]
25      pose = poses[images_idx]
26
27      #
28      #####
29
30      #                                     TODO
31      #
32      #
33      #####
34
35      # Perform training. Use mse loss for loss calculation and update the
36      # model parameter using the optimizer.
37      # Hint: focal is defined as a global variable in previous section
38      rays_o, rays_d = get_rays(H, W, focal=focal, pose=pose)
39      rgb, depth = render(model=model, rays_o=rays_o, rays_d=rays_d, near
40      =1., far=6., n_samples=n_samples, rand=True)
41
42      loss = torch.nn.functional.mse_loss(rgb, target)
43      optimizer.zero_grad()
44      loss.backward()
45      optimizer.step()
46
47      #
48      #####
49
50      #                                     END OF YOUR CODE
51      #
52      #
53      #####
54
55      if i % plot_step == 0:
56          torch.save(model.state_dict(), 'ckpt.pth')
57          # Render a test image to evaluate the current model performance.
58          with torch.no_grad():
59              rays_o, rays_d = get_rays(H, W, focal, testpose)
60              rgb, depth = render(model, rays_o, rays_d, near=2., far=6.,
61              n_samples=n_samples)
62              loss = torch.nn.functional.mse_loss(rgb, testing)
63              # Calculate PSNR for the rendered image.

```

```

52     psnr = mse2psnr(loss)
53
54     psnrs.append(psnr.detach().cpu().numpy())
55     iternums.append(i)
56
57     # Plotting the rendered image and PSNR over iterations.
58     plt.figure(figsize=(9, 3))
59
60     plt.subplot(131)
61     picture = rgb.cpu() # Copy the rendered image from GPU to CPU.
62     plt.imshow(picture)
63     plt.title(f'RGB Iter {i}')
64
65     plt.subplot(132)
66     picture = depth.cpu() * (rgb.cpu().mean(-1)>1e-2)
67     plt.imshow(picture, cmap='gray')
68     plt.title(f'Depth Iter {i}')
69
70     plt.subplot(133)
71     plt.plot(iternums, psnrs)
72     plt.title('PSNR')
73     plt.show()

```

## 2.4 Result

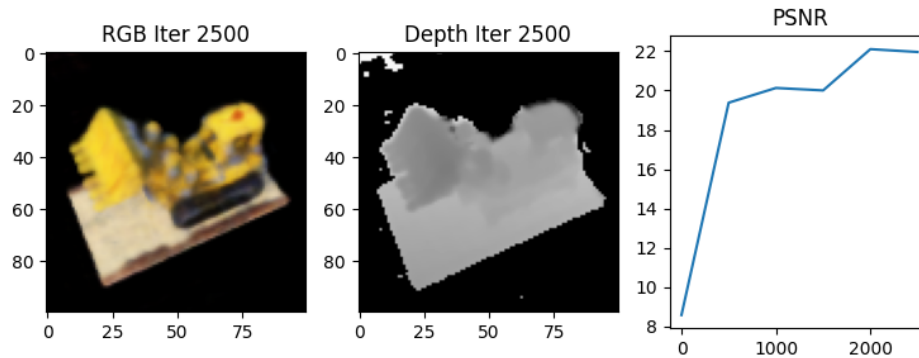


Figure 4: Training Result

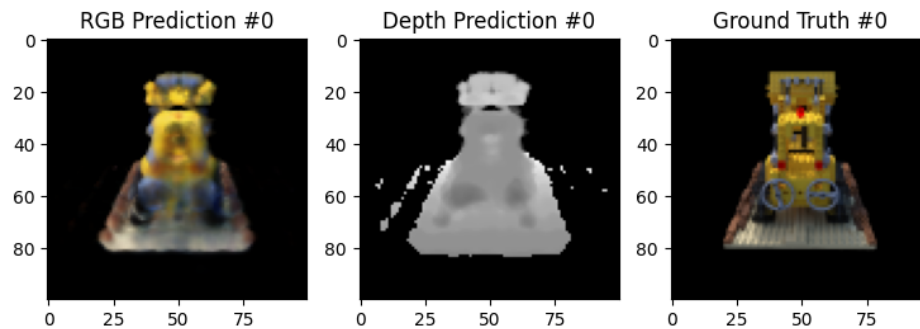


Figure 5: RGB Prediction #0

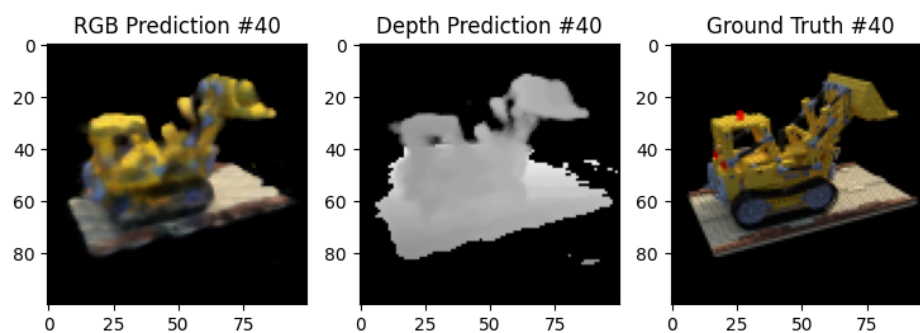


Figure 6: RGB Prediction #40

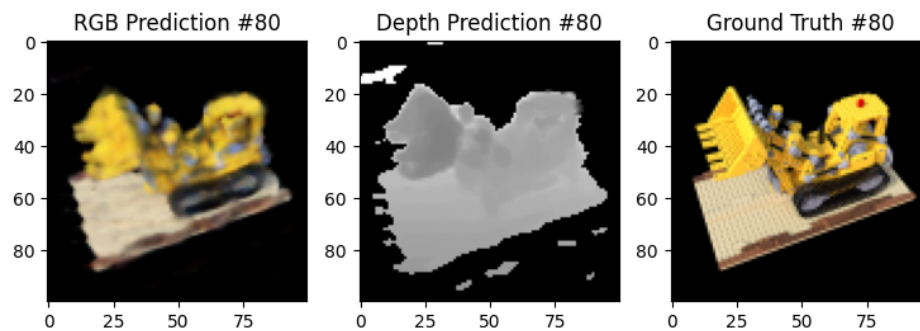


Figure 7: RGB Prediction #80

### 3 Appendix

Full Notebook pdf given in next page

*Submitted by Wensong Hu on April 17th, 2024.*