SRMCV Project: Extending the Diffeomorphic Neural Reconstruction Approach - Week 1 Report

1 Overview

The aim of the first week was to create the foundation of our project implementation for the future. This included the creation of a dataset using Blender as well as running the original training as well as adapting it to work with our data. We also added a way to render partial images for reduced processing requirements in future training using NeRFs. Our code is based on the previously released code by the authors of the paper [1], which can be found here.

2 Memory Requirements

One of the main obstacles we have run into so far, is the amount of VRAM required by the existing implementation. The authors of the paper don't specifically mention what hardware they used for training (not even in their supplemental material [1]), however from our tests running the existing demo code provided in their GitHub repository seems to require a GPU with 16GB of RAM.

3 Verification of the existing implementation

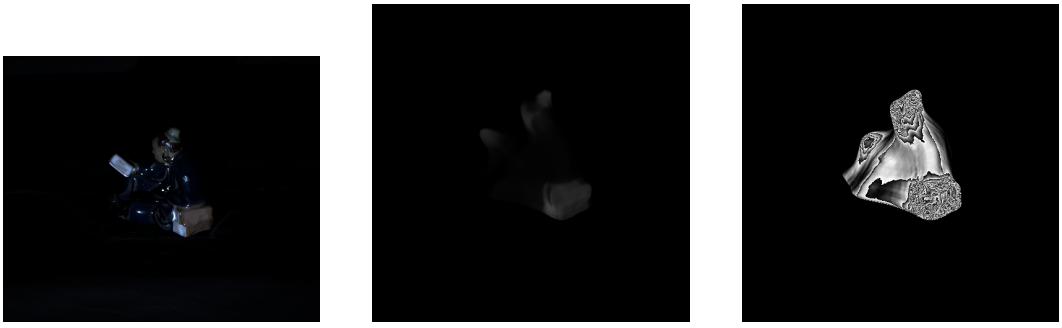

Figure 1: View of the object to be reconstructed, result of training after 120 iterations and after 85 iterations with a brightness multiplied by 100 for better visibility.

In order to verify the existing implementation we made some modifications to the existing dataloader and reduced the quality of the reconstruction by changing some of the hyperparameters (specifically changing the number of timesteps we integrate on the velocity field from 20 to 10). This allowed us to run the existing implementation on an Nvidida GTX 1070 (8GB VRAM). In the paper the training is run for 4500 epochs. A render of the original object, the result after 120 epochs (about 40 min of training) can be seen in Figure 1. The third image also shows the object after 85 iterations with the brightness multiplied by 100x for better visibility of the shape.

Once again, runtime is not something specifically discussed in the original paper. The only mention of training time is the statement that further work will be needed to reduce training time, implying a slow convergence.

4 Training on our dataset (A2)

We created a dataloader to allow our own dataset to be loaded and created a training script. One thing our dataset is however missing so far compared to the original dataset is a mask for each pose. The original implementation includes a loss based on the silhouette of the original. When training without this loss we find that the training does not converge but instead moves the mesh out of frame in order to reduce the loss. In other words the training currently does not work and we need to work out what exactly causes this issue.

5 Creating Dataset (A3)

In task A3, we had to implement a simple dataset given a mesh of a bunny by using Blender. In this project, we will consider only the head of the bunny, so we cut off the head of the bunny using the bisect tool and removed the part of the body. We recentered the head to the origin and scaled it until occupied a large portion of the unit sphere, while still being strictly inside. In this way, the convergence should be better since in the primary code, the geometry is initialized with the unit sphere. Then, we triangulated the new mesh, exported a new file, and removed the normal vectors of it.

Now, in order to create our new dataset, we used two cameras at different heights, and make them turn around the object, but only half of the full rotation and make sure that the unit sphere is always fully visible in the image plane as can be seen in figures 2 and 3. So with each camera, we took 30 different keyframes in Blender focusing only on the frontal part of the head of the new mesh, and used these camera settings to render 30 different images with soft Rasterizer. I used exactly the same soft Rasterizer as in assignment 4, where in the "SoftCookTorranceShader", we modified the function "_apply_lighting_cook_torrance" by excluding every variable related to specularity and replaced the diffuse_albedo with a tensor full of ones for the white color. We assume that exactly the same light is used for all the pictures, so we generated all the images using the same L0 value (L0=10.0), we divided the rendered images by the maximum brightness of all the images together and then saved them as 16bit png files.

It is worth mentioning that by using a low resolution (100x100) for the images we faced a problem where the rendered images contained some white patches. At the coarse stage, the rasterizer breaks the image into tiles (defined by bin_size) and records the faces that fall within each tile. The max_faces_per_bin is either defined by the user otherwise it's computed with a heuristic, and if there is more faces per bin than the number specified, our rasterizer overflows and thus causes the white patches. So we had to set the max_faces_per_bin in the rasterization settings in a higher value to solve our problem.

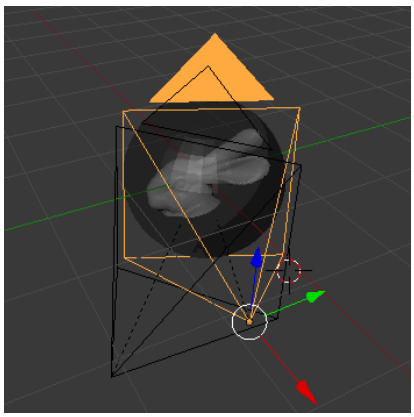

Figure 2: Cameras at different height turn around the object.

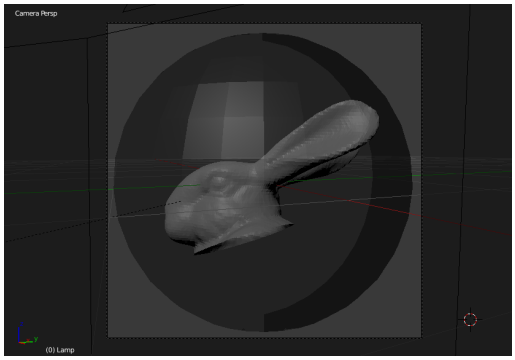

Figure 3: The unit sphere is always fully visible in the image plane, while the head occupies a large portion of it.

6 Rendering a Subpart of the Image (B)

The final task for the week was to implement code to make it possible to render a random subpart of an image at each iteration of training. This is to lighten the computational load for the future when we will use NeRF. [2].

The implementation can be found under *'utils.py'* and the *'random_crop'* function. The function parameters are the camera translation matrix T, image_size to be rendered, and the crop_ratio. Before the training, the configuration parameters for cropping is updated to 'True', if rendering subpart of an image is desired. This parameter is defined in 'rendering.rgb.crop'. Moreover, if that is set to 'True', the crop ratio hyperparameter is set to 'rendering.rgb.crop_ratio'.

Here, the crop ratio is a hyperparameter that decides which ratio of the image will be rendered. The default value is 0.5. For instance, for an image of 500x500, if we want to render only a 250x250 subpart, the crop ratio is set to be 0.5.

Please note that, the crop ratio is used to set how close the camera should be to the mesh. This is done by using the z-axis information of the translation matrix (T) of the camera. T is updated by multiplying the z-axis of the T matrix with the crop ratio.

After setting the crop ratio, during each iteration, random values for x and y coordinates of T are assigned to translate the camera on the xy-plane. This helps to choose a random area of the image at each iteration. Finally, the image size and the new T matrix of the camera is updated using the new x-y coordinates and the crop ratio.

Finally, note that, this code could not be tested on our pipeline due to insufficient GPU memory. However, it is tested on default Pytorch3D dataset provided in this official demo. The results can be seen in Figure 4.

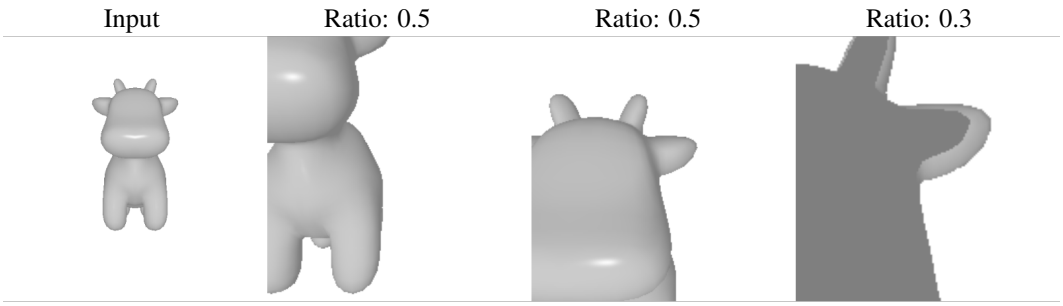

Figure 4: Results of different crops. The second and third images use the same crop ratio with different camera xy-coordinates. This is due to randomized translation on the xy-plane.

7 Next Steps

While our rendering approach works successfully, we failed to run the whole training on our computers or the servers provided by TUM due to lack of sufficient GPU memory (VRAM). To overcome this challenge, we asked whether it is possible to use a more powerful computer resource provided by TUM. If that fails, we can try other options such as Amazon services or Google Colab Pro. GPU power is crucial for this project, since we will likely require even higher computational power after introducing NeRF. After solving this challenge, we can move on with the other tasks.

8 Task assignment

Vasiliki Papadouli:

- Creation of dataset using Blender and Pytorch3D's differentiable rendering

Alexander Fuchs:

- Modification of original code for lower memory requirements

- Modification of dataloader and training routine for our dataset
- Some Debugging help for differentiable rendering

Zehranaz Canfes:

- Implementation of random cropping function

References

[1] Ziang Cheng, Hongdong Li, Richard Hartley, Yinqiang Zheng, and Imari Sato. Diffeomorphic neural surface parameterization for 3d and reflectance acquisition. In *ACM SIGGRAPH 2022 Conference Proceedings*, pages 1–10, 2022. 1

[2] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021. 3