Unraveling Imagination Synthesis

constructing Conceptual Artistry into a 3D scene through Neural Radiance Fields

(David) Zichen Liu[a]

[a]*Shenzhen college of international education,*

Abstract

In this research, I propose an architecture that can reconstructing images into implicit 3D representation through Neural radiance field (NeRF). With a consistent combination between the world graphics and computer vision, I dive deep into the detailed implementation of such model and discovers the constrain in time usage as well as computational power, I strongly believe that in the future, this area will become the pioneering research in meta verse, virtue reality and augmented reality.

Keywords: Graphics, 3D reconstruction, computer vision, NeRF

1. Introduction

Realistic or Imaginary 3D scene construction have always been a difficult problem to achieve, however, highly demanded in the fields of gaming, virtual reality, and robotic receptions. Imagine that We can obtain the novel 3d structure just with the 2D images whether if it's photo from reality or conceptual art made by artist? In order to give the computer the power of understanding 3D constitution from 2D images, we can empower it with the advanced methods – Neural radiance Field (NeRF) (Mildenhall et al. 2021),(Gao et al. 2022) which proposing a state-of-the-art result in synthesizing novel views based on a single or Multi-view image. Extended the new possibility for machines to understand from a visual perspective. To summarize what I have done for the scene construction, the first stage is to train a generative model that's based on NeRF representation of light fields, with a rendering process mimic by an Multi Layer Perceptron (MLP), which provide us with differentiable feed-forward process that can be backpropagate with loss from image.

As NeRF use a process of reverse the common rendering process of geometry to image pipeline, which can be classified as a way of differentiable rendering (Kato et al. 2020). In a classical rendering process Rasterization (Pineda 1988) and Ray-tracing (Appel 1968) are commonly used algorithm that samples image first from meshes in a certain camera view to obtain the transformation coordinate with its experience called the vertex shader, and then go through a rasterizer and fragment shader to obtain the final image. For the purpose of reverse the whole process, I have to make sure every step within the rendering process is differentiable therefore I am able to apply chain role to backpropagate images back to 3D representation. In the rendering process, all the coordinate transformation can be defined by matrix multination and the texturing process are defined by the user thus they are differentiable, while in the rasterization process the color need to be considered as a constant, although I might know that if there is a great shift between the color value, this might indicate an edge of the 3D representation I intended to reconstruct, therefore the whole process is not reversible. One way of approximating the gradient is to consider the image as a field of color (Loper and Black 2014) otherwise there is also some methods that rebuild the whole rendering process with each stage differentiable (Liu et al. 2019) for explicit 3D representation like mesh. NeRF on the other hand, stimulate this process by using an MLP (although it's have been proven that neural network is not necessary (Fridovich-Keil et al. 2022)) to mimic the implicit representation function with an input of 5D coordinate and output a voxel scene (the detail explanation of how it functions please refer 3.1).

For an imaginary scene created by artist, it's impossible to obtain the camera position which are crucial to NeRF, compare to real picture capture by the camera. Therefore, I need to estimate the camera status from the drawing beforehand. To accomplish this, visual optometry that estimate the transformation between cameras

based on the images that I captured.

In this research, I produce a pipeline of generating both realistic and imagery content that are represented in 3D by using NeRF with camera pose estimation. Producing a next generation workflow that might empowers many areas include VR/AR, conceptual art, or even advertisement.

2. Literature review

2.1. Text Guided/shape awareness Object Generation

It's essential to be aware that to generate a 3D representation from art work, there is no shapes and multi-view data to optimize the Neural radiance field. Text alignment to 3D shapes in NeRF are popular researching field (Jain et al. 2022)(Poole et al. 2022) (Lin et al. 2023) as it provides more flexibility and modifiability for human user to interact with while nature language input is considering been easier to use compare to drawing. DreamField (Jain et al. 2022) provide a simple and constrained 3D representation with neural guidance that supports diverse generation from caption zero-shot, the text-alignment process is achieve by pretrain representation using CLIP (Radford et al. 2021) which consist of transformer image encoder with a masked transformer text encoder. Further more Dreamfusion (Poole et al. 2022) empower its process with the methods of stable diffusion model (Rombach et al. 2022). While providing a shape awareness methods with another approach with stochastic process over Variational Auto Encoder.

2.2. Scene-scale Contraction

Enable to model the 2D representation into a large-scale 3D scene representation instead of object centric reconstruction in order to understand the special relationship of object in space. Block-Nerf (Tancik et al. 2022), decompose the scene into individual trained NeRFs to represent a large-scale environments. Still facing the difficulties in presenting transient object like cars, lack of connection between each block, and constrain by the computational power. While Neuralangelo (Li et al. 2023), achieve the fine details for a large scene using a coarse-to-fine optimization by fine to the hyper-parameter within training the MLP for NeRF, preventing over-fit to local minimum. Since egocentric scene is highly required in the robotics scenario, map reconstruction is also a significant process in understanding the environment. Nice-SLAM (ibid.) a simultaneous localization and mapping (SLAM) (Fuentes-Pacheco, Ruiz-Ascencio, and Rendón-Mancha 2015) algorithm, which introduce a hierarchical reconstruction methods involve a pretrained differentiable renderer from coarse to fine level feature grid with Mean Square Error (MSE) loss within depth and image.

2.3. Reconstruction without pose prior

To train a differentiable rendering frame work, a camera poses is usually a curtail component that contribute to matrix transformation and bridging image with scene. However, when dealing with conceptual artwork, I barely have the ability to obtain the camera poses for a drawing. Therefore, building the scene from a prior without a pose is the main challenge here. The most common way to optimize pose is by self-supervised learning, where researcher can append posture optimization into NeRF's volume rendering process. Where iNeRF (Yen-Chen et al. 2021) optimize pose within the gradient flow, updating the hyperparameters during backpropagation. Although the camera pose calculation in NeRF is non-linear, as it involve affine transformation matrix multiplication, iNeRF addresses this by replacing the matrix with Lie algebra, changing multiplication to variable addition which allows gradient decent applied. For complex trajectory estimation, Nope-NeRF (Bian et al. 2023) demonstrate using mono-depth maps to optimize NeRF, regularize geometry from sequence of images with massive amount of motion between cameras.

3. 3 Methodology

3.1. NeRF

3.1.1. Scene in Neural radiance field

NeRF representing scene implicitly by a mapping $F_\theta(x, d) \rightarrow (c, \sigma)$ which input special location $x \in \mathbb{R}^3$ with special Cartesian unit vector d replacing viewing direction (θ, ϕ) in application and output RGB color c with it's volume density σ.

3.1.2. Volume rendering

Volume rendering are a ray tracing based algorithm that core in NeRF, rendering image both used in the calculating the optimisation loss and synthesis finial result. (Kajiya and Von Herzen 1984) The intensity (color intensity) of light along a ray can be given by a function $L(r(t), d)$ for $r(t) = o + t * d$ set of particle on the ray Which change in L after the light transmit through a unit volume along direction d is given by $emission + inscattering–absorption$ In NeRF the scene is been modeled as a cloud of space with particle emitting light whose intensity change though out the motion of photon. In practical I only use the absorption

plus emission to done the volume rendering process to reduce massive calculation done on scattering light between particles. (However, this leads to a constrain that my scene can only stay on the original lighting condition, without the ability to modify as well as combine multiple scene. (Zheng, Singh, and Seidel 2021))

For a space given front position t_n to back position t_f, lets assume s to be the distance from the front not overpassing t_f along the ray. With $\sigma(s)$ as the volume density, defined as probability density of photon been absorb by the space, with reciprocal unit of length. Therefore it's easy to derive that for a unit length of δs at a given position of s, the probability of the light been absorb is $\delta s * \sigma(s)$. So the intensity of the light decrease along a minimal change with function of

$$L(s + \delta s) = L(s) * (1 - \delta s * \sigma(s)) \tag{1}$$

Reorganize the formula to rate of change in intensity gives

$$\frac{L(s + \delta s) - L(s)}{\delta s} = -L(s) * \sigma(s)$$
$$L(s)' = -L(s) * \sigma(s)$$

Solving this Ordinary Differential Equation ODE gives an expression of $L(s)$

$$L(s) = e^{-\int_{n=0}^{s} \sigma(x)\delta x} \tag{2}$$

Defining the transparency along a give distance s to s', as $T = \frac{L_o}{L_i}$ which L_o indicate the input intensity and L_o as output. Having this, the RGB value gained after rendering can be calculate by adding up color contribute by each particle along the ray.

$$C(r) = \int_{t_n}^{t_f)} T(s) * \sigma(r(s) * c(r(s), d)\delta s \tag{3}$$

To calculate this in practice, the sample position s should be discrete to

$$s_i = U[t_n + \frac{i-1}{N} * (t_f - t_n), t_n + \frac{i}{N} * (t_f - t_n)] \tag{4}$$

thus estimate the color with quadrature rule, $\delta_i = s_i + 1 - s_i$, with $\alpha_i = 1 - e^{\sigma_i * \delta_i}$ be the density of color in the given length (Max 1995)

$$T_i = e^{-\sum_1^{i-1} \sigma_j * \delta_j} \tag{5}$$

$$C'(r) = \sum_{i=1}^{N} T_i * \alpha_i * c_i \tag{6}$$

If you interest in detail explanation, please have a look at further conclude derivation. (Tagliasacchi and Mildenhall 2022)

3.2. Optimize MLP

With the design of the general structure, however, there is still problems of not been able for the MLP to fit a fine-grained detail as well as the computational inefficiency. NeRF introduce two methods following to address the two problems.

3.2.1. Positional encoding

Recent studies had shown that Polynomial Neural Networks (PNN) has a specific bias over the low frequency functions, which shows a slower learning rate in high frequency domain. The 5-dimensional input for NeRF is not able to train the detail information efficiently from the image. However, mapping the input into high dimensional space using high frequency function which allows it to fit the fine-grained information within the data. Intuitively NeRF modify the original MLP F'_θ with a encoding function γ represent R to R^{2L}

$$\gamma(p) = (sin(2^{0*\pi*p}), \ldots, cos(2^{(L-1)*\pi*p})) \tag{7}$$

$$F_\theta = F'_\theta \circ \gamma \tag{8}$$

Which gives the input Fourier feature with location normalize to $[-1, 1]$, thus in an iteration with same loss propagate back to hyper-parameters, the high frequency part contribute a much more massive weight in the final outcome, which allows learning the much finer details.

3.2.2. Hierarchical volume sampling

Since it's time-consuming to direct sample along the ray using N step. It's better to strategies the sampling process from coarse to fine. Which propose a hierarchical way sampling along the ray. In coarse sampling stage, I nomalize the alpha value to obtain a piecewise-constant Probability Density Function (PDF) of which part along the ray contribute more on the finial rendered image.

$$\alpha'_i = \frac{\alpha_i}{\sum_{j=1}^{N_c} \alpha_j} \tag{9}$$

applying inverse transform sampling, by sample's a piecewise-constant set along the range to get a new set

of sample point that obey the distribution of the PDF, which than added with the first set of sample point to perform a second round of rendering.

3.3. Camera Calibration

To work out the rotation and translation of a camera from the 2D images is usually defined as a pose-estimation problem which is fairly important and well-studied in the world of graphics, computer vision and robotics. To simplified the calculation and considering the most common cases, my implementation only stick on the pinhole camera model for simplification, if you are interested in different camera model, please refer to the following passage (Sturm et al. 2011).

3.3.1. Camera and Image

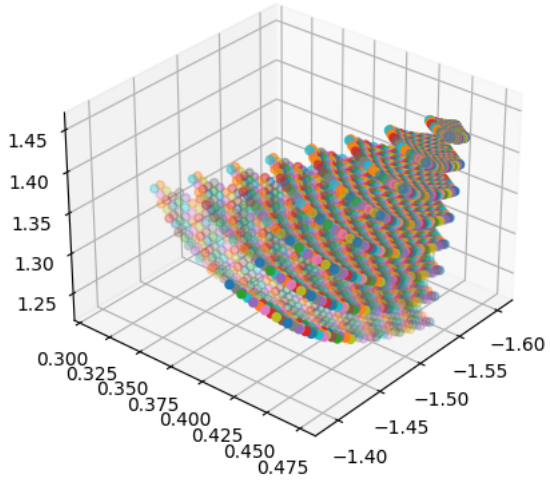

Figure 1: Light ray sample from a pinhole camera model

Pinhole camera model is the most simple and straight forward camera model used for a mono camera which is developed based on the physics phenomenon of candle's image appear on a plane at the other side of a pin-hole. The model consider light as a straight ray, allowing us to calculate the translation relation between the pixel (u, v) from image coordinate of the image, from the camera intrinsic matrix K which define based on its internal parameter include focal lens f, and the extrinsic matrix $E \in \mathbb{R}^{4\times4}$ which describe the rotation R and translation t of the camera in the world coordinate. Given by following equation:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{z} \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{10}$$

3.3.2. Visual Odometry (VO)

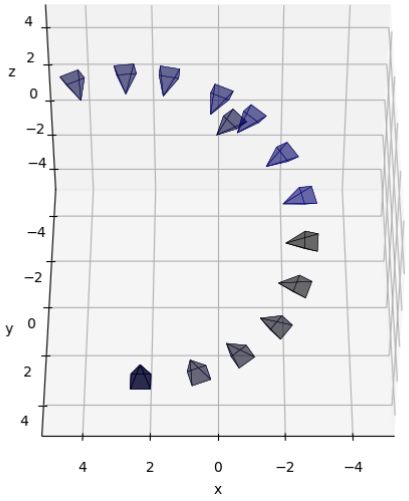

Figure 2: Visualisation of trajectory of multiple camera from circular path

Visual Odometry general term referring to the methods used in motion estimation based on the image taken by the sensor though out the time domain. In order to achieve a concordant result in poses estimation, I must produce a solid way in evaluating it. When a organism is locating it's direction in a map, it usually considering the points that features the location and recognizable after movement. Similarly, VO done this by comparing the chosen feature point, consist of Key-point and Descriptor (a vector designed to describe the information around the chosen pixel). By cleverly choosing and pairing those point in continues frame, it's then capable to find the motion of the camera. A common implemented algorithm is Oriented FAST and Rotated BRIEF (Aglave and Kolkure 2015)

To minimize the lose design for the pairing point, 2D-2D: Epipolar Geometry (Larsson, Pollefeys, and Oskarsson 2021) is used to give out a linear equation of translation follow by epipolar constraint, essential Matrix $E' = t^{\wedge} * R$ decomposed by to it's singular value (SVD) (Kalman 1996) which leave a terms containing E' and produce least squared methods or Random Sample Consensus (RANSAC) (Bahraini, Bozorg, and Rad

2018) to optimize it. With all this been done, I am able to apply the classical methods of Triangulation (Hartley and Sturm 1997) to calibrate my camera with depth from the images.

4. Experiment

4.1. Scope estimation and difficulty indication

The central target of this research is the development and utilization of the Neural radiance field, a state-of-the-art technique in computer vision and world of machine learning used in novel scene synthesis and 3D object generation. Thus, I need to investigating methods to enhance the performance, robustness and scalability of NeRF. One primarily difficulty lies in the optimizing the NeRF algorithm, in other word a clever design of the training pipeline and the generation process. Base on my experiment on other's methods in generating high resolution object (Lin et al. 2023) (referring to 4.2) can be time-consuming as it involves numerous iterations to reach desired outcomes. training a NeRF model in a 3D scene can take up super long time (10 days in one RTX-3080) and gives a poor result. So, I should carefully choose the back bone I an using and aim in a minimal scale. Collecting and curating 2D conceptual art for conversion into 3D scenes can be a labor-intensive process. It may involve digitizing physical artwork, finding high-quality digital images, or creating artwork from scratch, which can consume a significant amount of time and effort. While there might take time to apply a robustness backbone and transfer it onto the scene of Art-2-3D task

4.2. Implementation

```
/* file structure */
|-- config file #setup parameter
        |-- get config.py
        |-- config.txt
|-- data
        |-- input #data used for training and testing
                |-- xxx.jpg
                |-- xxx.ply
        |-- output #outputed images
                |-- xxx.jpg
|-- src #the helper function which defined as the back end of the program
        |-- Utils # helper libraries like colmap
        |-- visualization # visualization of camera
        |-- Nerf.py
        |-- rendering.py #function used in volume rendering
|-- train.py #running file to run the file
```

Figure 3: General file structure of the program

my frame work was mostly developed using python programming language and pytorch frame work in building neural network and implementing tensor operation based on linear algebra. For the purpose of convenient dependencies management and storage. This early-stage experiment are all test on Google-Colab and Linux Ubuntu 22.04 release, which provide CUDA environment. The training stage involve the use of T4 GPU, V100 GPU Nvidia provided by colab and RTX3080 Nvidia on Ubuntu. The core NeRF structure is based on the original design with 11 hidden layers with dimension 128 and reduce by halve for the last two. Activation function (Sagar Sharma, Simone Sharma, and Athaiya 2017) of ReLU is used between each layer, and Sigmoid for the last output RGB value. An extra dim is given at 9th layer to give out the density (sigma) before head which allow the front layer focus on finding the geometrical distribution and last 2 on colors and texture. Besides the uses of the human synthetic data "Tractorn" provided by the original NeRF which data is well distributed around the hemisphere of the parametrized 3D model. I further trained the model using real-world images collect by easy access mobile device. The images of individual object are taken using a circular trajectory around it

Figure 4: User input image

which helps to eliminate the scale ambiguity in the mono camera by providing translation, In addition capture more information from the object. As long as I gather the images, Colmap, a general implementation to solve the problem of Structure From Motion (SFM) (Özyeşil et al. 2017) as well as the Multi-View Stereo (MVS) (Furukawa, Hernández, et al. 2015), is used as a pre-built wheel to estimate the camera poses and in-

ternal parameter. The data are sampled as a field of light rays radiate out from individual cameras by a transformation matrix from pixel coordinate to $\mathbb{R}^3$ structure space. Each ray are presented as a collaboration of ray origins ($\mathbb{R}^3$), ray directions ($\mathbb{R}^3$, vector form), and the pixel value as ground truth ($\mathbb{R}^3$). In addition, to maintain the consistency of gradient in back propagation stage, the input value should be normalized.

5. Discussion

5.1. Result and Evaluation

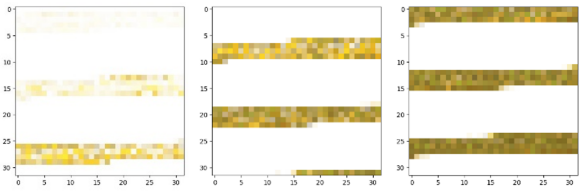

Figure 5: Failure case with lack in training data

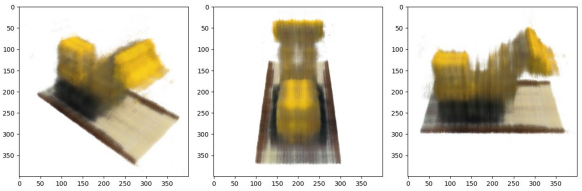

Figure 6: Course sampling example

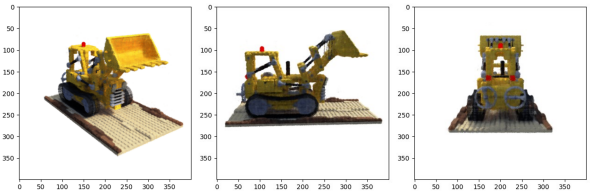

Figure 7: Hierarchical sampling example

The first experience are tested on human synthetic dataset used in the original implication of NeRF model. With the upper three picture showing the result with a failure case with a minimum number of light rays, while the middle one shows the course rendering stage without empowering by the hierarchical sampling strategy. The bottom third shows the rendering result with one more epoch of hierarchical sampling which shows a detailed generated novel scene.

Next I test my model on realistic photo taken though normal live mobile device, the first row shows failure

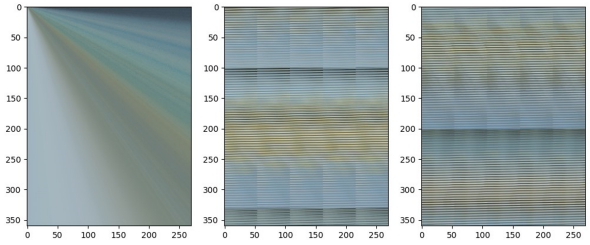

Figure 8: Failure case with wrong coordinate and lack in training data

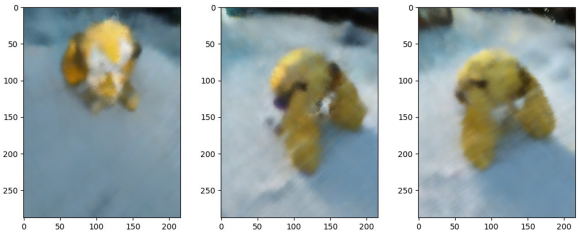

Figure 9: Finial results

cases with wrong calculation pixel to world sampling and not sufficient amount of data. While the last row demonstrate the capability of NeRF in rendering novel view as well as edit color though it's implicit representation.

5.2. Limitation

The sample algorithm calculate 3 for-loop iteratively. The number of process is calculated by num_Of_Camera * H * W which produce low efficiency in data sampling when the inputs is getting larger (current cases: 15 * 1420 * 1080 > 1e07). To imply the methods onto real world cases, and process in real time, there is clearly a needs for further algorithm design, more Computation power or even a customized hardware, and microprocessor.

5.3. Conclusion

In this work, I build an architecture that allow it's user to upgrade a picture taken from the real world into synthetic 3d spaces without geometrical prior given. despite that it's now stays in a prototype state without a user friendly interface and highly constrain in computational power and time usage. I believe that in the future, with time consistence diffusion workflow been implemented in, we will be able to construct 3d structure from simply your imagination, which shows a great potential in the fields of VR, AR and meta verse. Application

involves producing interactive collections in cloud museum; Creating digital class room, favor the student in young age or learning design and architecture; Providing a new methods and tool set for artiest to explore the world.

References

Aglave, Prashant and Vijaykumar S Kolkure (2015). "Implementation Of High Performance Feature Extraction Method Using Oriented Fast And Rotated Brief Algorithm". In: *Int. J. Res. Eng. Technol* 4, pp. 394–397.

Appel, Arthur (1968). "Some techniques for shading machine renderings of solids". In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pp. 37–45.

Bahraini, Masoud S, Mohammad Bozorg, and Ahmad B Rad (2018). "SLAM in dynamic environments via ML-RANSAC". In: *Mechatronics* 49, pp. 105–118.

Bian, Wenjing et al. (2023). "Nope-nerf: Optimising neural radiance field with no pose prior". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4160–4169.

Fridovich-Keil, Sara et al. (2022). "Plenoxels: Radiance fields without neural networks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5501–5510.

Fuentes-Pacheco, Jorge, José Ruiz-Ascencio, and Juan Manuel Rendón-Mancha (2015). "Visual simultaneous localization and mapping: a survey". In: *Artificial intelligence review* 43, pp. 55–81.

Furukawa, Yasutaka, Carlos Hernández, et al. (2015). "Multi-view stereo: A tutorial". In: *Foundations and Trends® in Computer Graphics and Vision* 9.1-2, pp. 1–148.

Gao, Kyle et al. (2022). "Nerf: Neural radiance field in 3d vision, a comprehensive review". In: *arXiv preprint arXiv:2210.00379*.

Hartley, Richard I and Peter Sturm (1997). "Triangulation". In: *Computer vision and image understanding* 68.2, pp. 146–157.

Jain, Ajay et al. (2022). "Zero-shot text-guided object generation with dream fields". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 867–876.

Kajiya, James T and Brian P Von Herzen (1984). "Ray tracing volume densities". In: *ACM SIGGRAPH computer graphics* 18.3, pp. 165–174.

Kalman, Dan (1996). "A singularly valuable decomposition: the SVD of a matrix". In: *The college mathematics journal* 27.1, pp. 2–23.

Kato, Hiroharu et al. (2020). "Differentiable rendering: A survey". In: *arXiv preprint arXiv:2006.12057*.

Larsson, Viktor, Marc Pollefeys, and Magnus Oskarsson (2021). "Orthographic-perspective epipolar geometry". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 5570–5578.

Li, Zhaoshuo et al. (2023). "Neuralangelo: High-fidelity neural surface reconstruction". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8456–8465.

Lin, Chen-Hsuan et al. (2023). "Magic3d: High-resolution text-to-3d content creation". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 300–309.

Liu, Shichen et al. (2019). "Soft rasterizer: A differentiable renderer for image-based 3d reasoning". In: *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 7708–7717.

Loper, Matthew M and Michael J Black (2014). "OpenDR: An approximate differentiable renderer". In: *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part VII 13*. Springer, pp. 154–169.

Max, Nelson (1995). "Optical models for direct volume rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 1.2, pp. 99–108.

Mildenhall, Ben et al. (2021). "Nerf: Representing scenes as neural radiance fields for view synthesis". In: *Communications of the ACM* 65.1, pp. 99–106.

Özyeşil, Onur et al. (2017). "A survey of structure from motion*." In: *Acta Numerica* 26, pp. 305–364.

Pineda, Juan (1988). "A parallel algorithm for polygon rasterization". In: *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pp. 17–20.

Poole, Ben et al. (2022). "Dreamfusion: Text-to-3d using 2d diffusion". In: *arXiv preprint arXiv:2209.14988*.

Radford, Alec et al. (2021). "Learning transferable visual models from natural language supervision". In: *International conference on machine learning*. PMLR, pp. 8748–8763.

Rombach, Robin et al. (2022). "High-resolution image synthesis with latent diffusion models". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10684–10695.

Sharma, Sagar, Simone Sharma, and Anidhya Athaiya (2017). "Activation functions in neural networks". In: *Towards Data Sci* 6.12, pp. 310–316.

Sturm, Peter et al. (2011). "Camera models and fundamental concepts used in geometric computer vision". In: *Foundations and Trends® in Computer Graphics and Vision* 6.1–2, pp. 1–183.

Tagliasacchi, Andrea and Ben Mildenhall (2022). "Volume rendering digest (for nerf)". In: *arXiv preprint arXiv:2209.02417*.

Tancik, Matthew et al. (2022). "Block-nerf: Scalable large scene neural view synthesis". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8248–8258.

Yen-Chen, Lin et al. (2021). "inerf: Inverting neural radiance fields for pose estimation". In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 1323–1330.

Zheng, Quan, Gurprit Singh, and Hans-Peter Seidel (2021). "Neural relightable participating media rendering". In: *Advances in Neural Information Processing Systems* 34, pp. 15203–15215.

6. Appendix

Code

The main training loop of the program (train.py):

```
import torch
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader

from configs.get_configs import get_args
#from datasets import dataset_dict

from srcs.Nerf import NerfModel
from srcs.render import render_rays, compute_accumulated_transmittance

def train(nerf_model, optimizer, scheduler, data_loader, device='cpu', hn=0, hf=1,
    nb_epochs=int(1e5),
            nb_bins=192, H=400, W=400, fine_stage=2):

  training_loss = []

  for _ in tqdm(range(nb_epochs)):
    for batch in data_loader:
      #for each training point in the batch it exist in form [origin(x,y,z), direction(a1,
    a2,a3), gt_pixel(r,g,b)]
      ray_origins = batch[:,:3].to(device)
      ray_directions = batch[:,3:6].to(device)
      ground_truth_px_values = batch[:,6:].to(device)
      if _ < fine_stage:
        regenerated_px_values = render_rays(nerf_model, ray_origins, ray_directions, hn=hn,
     hf=hf, nb_bins=nb_bins, stage='coarse')
      else:
        regenerated_px_values = render_rays(nerf_model, ray_origins, ray_directions, hn=hn,
     hf=hf, nb_bins=nb_bins, stage='fine')
      loss = ((ground_truth_px_values - regenerated_px_values) ** 2).sum()

      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
      training_loss.append(loss.item())
    scheduler.step()
    for img_index in range(3):
      test(hn, hf, testing_dataset, img_index=img_index, nb_bins=nb_bins, H=H, W=W)

  return training_loss

@torch.no_grad()
def test(hn, hf, dataset, chunk_size=5, img_index=0, nb_bins=192, H=400,W=400):
  ray_origins = dataset[img_index * H * W: (img_index+1)* H * W, :3]
  ray_directions = dataset[img_index * H * W: (img_index+1)* H * W, 3:6]

  data = []

  for i in range(int(np.ceil(H / chunk_size))):

    ray_origins_ = ray_origins[i * W * chunk_size : (i+1) * W * chunk_size].to(device)
    ray_directions_ = ray_directions[i * W * chunk_size : (i+1) * W * chunk_size].to(device
    )
    regenerated_px_values = render_rays(model, ray_origins_, ray_directions_, hn=hn, hf=hf,
     nb_bins = nb_bins)
    #print("rpv:",regenerated_px_values)
    data.append(regenerated_px_values)
  img = torch.cat(data).data.cpu().numpy().reshape(H,W,3)
```

```
  plt.figure()
  plt.imshow(img)
  plt.savefig(f'output/novel_views_2/img_{img_index}.png', bbox_inches='tight')
  plt.close()

if __name__ == '__main__':
  # get training argument
  args = get_args()
  device = args.device

  # get dataset
  # todo - load data to correct format [img.jpg + poses + ] -> [origin + direction + ground_truth] for training
  training_dataset = torch.from_numpy(np.load('datasets/fufu/short_cut/training_data.pkl', allow_pickle=True))
  testing_dataset = torch.from_numpy(np.load('datasets/fufu/short_cut/training_data.pkl', allow_pickle=True))


  idx = torch.randperm(training_dataset.size(0))
  training_dataset = training_dataset[torch.sort(idx).indices,:]
  num_data_use = int(int(idx.size(0))/10)

  #training_dataset = training_dataset[:num_data_use,:]
  #print(testing_dataset)

  # initialize NeRF
  model = NerfModel(hidden_dim=256).to(device)
  model_optimizer = torch.optim.Adam(model.parameters(), lr=5e-4)
  scheduler = torch.optim.lr_scheduler.MultiStepLR(model_optimizer, milestones=[2, 4, 8], gamma=0.5)

  data_loader = DataLoader(training_dataset, batch_size=16, shuffle=True) # fit data to batches

  train(model, model_optimizer, scheduler, data_loader, nb_epochs=args.num_epochs, device=device, hn=2, hf=6, nb_bins=100, H=288, W=216, fine_stage = args.fine_stage)
```

NeRF architecture (Nerf.py):

```
import torch
import torch.nn as nn

class NerfModel(nn.Module):
  def __init__(self, embedding_dim_pos=10, embedding_dim_direction=4, hidden_dim=128):
    super(NerfModel, self).__init__()

    self.block1 = nn.Sequential(nn.Linear(6*embedding_dim_pos+3,hidden_dim), nn.ReLU(), # including original pos
                                nn.Linear(hidden_dim,hidden_dim), nn.ReLU(),
                                nn.Linear(hidden_dim,hidden_dim), nn.ReLU(),
                                nn.Linear(hidden_dim,hidden_dim), nn.ReLU(),)
    self.block2 = nn.Sequential(nn.Linear(6*embedding_dim_pos+hidden_dim+3,hidden_dim), nn.ReLU(),
                                nn.Linear(hidden_dim,hidden_dim), nn.ReLU(),
                                nn.Linear(hidden_dim,hidden_dim), nn.ReLU(),
                                nn.Linear(hidden_dim,hidden_dim+1), )
    self.block3 = nn.Sequential(nn.Linear(6*embedding_dim_direction+hidden_dim+3,hidden_dim//2), nn.ReLU(),)
    self.block4 = nn.Sequential(nn.Linear(hidden_dim // 2,3), nn.Sigmoid(),)

    self.embedding_dim_pos = embedding_dim_pos
    self.embedding_dim_direction = embedding_dim_direction
    self.relu = nn.ReLU()
```

```
  @staticmethod
  def positional_encoding(x, L):
    out = [x]
    for j in range(L):
      out.append(torch.sin(2**j * x))
      out.append(torch.cos(2**j * x))
    return torch.cat(out, dim=1)

  def forward(self, o, d):
    emb_x = self.positional_encoding(o, self.embedding_dim_pos)
    emb_d = self.positional_encoding(d, self.embedding_dim_direction)
    h = self.block1(emb_x)
    tmp = self.block2(torch.cat((h,emb_x),dim=1))
    h, sigma = tmp[:,:-1], self.relu(tmp[:,-1])
    h = self.block3(torch.cat((h,emb_d),dim=1))
    c = self.block4(h)
    return c,sigma
```

rendering stage (render.py):

```
    import torch
# utils
###
def compute_accumulated_transmittance(alphas):
  accumulated_transmittance = torch.cumprod(alphas,1) #cumprod get the cumulative product
    of alphas
  return torch.cat((torch.ones((accumulated_transmittance.shape[0],1), device=alphas.device
    ),
                    accumulated_transmittance[:,:-1]), dim = -1)

def PDF_reverse(sigma, hn, hf):
  # x_axis: sigma [nb_samples, nb_bins]
  device = sigma.device
  nb_samples = sigma.shape[0]
  nb_bins = sigma.shape[1]

  # normalization
  sigma = torch.cumsum(sigma,-1)
  n_sigma = sigma.div((1e-10)+sigma.sum(-1).unsqueeze(-1))


  t = torch.linspace(0, hf-hn, nb_bins, device=device).expand(nb_samples, nb_bins)
  off = torch.linspace(hn, hn, nb_bins, device=device).expand(nb_samples, nb_bins)


  t = t.mul(n_sigma)+off

  return t

def coarse_sample(ray_origins, ray_directions, hn=0, hf=0.5, nb_bins=192, nb_samples=100):
  device = ray_origins.device

  t = torch.linspace(hn,hf,nb_bins, device=device).expand(nb_samples, nb_bins)
  mid = (t[:,:-1]+t[:,1:]) / 2
  lower = torch.cat((t[:,:1],mid), -1)
  upper = torch.cat((mid, t[:,-1:]),-1)
  u = torch.rand(t.shape, device=device)
  t = lower + (upper - lower) * u


  delta = torch.cat((t[:,1:] - t[:,:-1], torch.tensor([1e10], device=device).expand(
    nb_samples, 1)),-1)

  x = ray_origins.unsqueeze(1) + t.unsqueeze(2) * ray_directions.unsqueeze(1)
```

```
  ray_directions = ray_directions.expand(nb_bins, ray_directions.shape[0], 3).transpose(0,1)

  return x, ray_directions, delta

def fine_sample(sigma, ray_origins, ray_directions, hn=0, hf=0.5, nb_bins=192, nb_samples=100):
  device = ray_origins.device

  t = PDF_reverse(sigma, hn, hf)

  mid = (t[:,:-1]+t[:,1:]) / 2
  lower = torch.cat((t[:,:1],mid), -1)
  upper = torch.cat((mid, t[:,-1:]),-1)
  u = torch.rand(t.shape, device=device)
  t = lower + (upper - lower) * u

  delta = torch.cat((t[:,1:] - t[:,:-1], torch.tensor([1e10], device=device).expand(nb_samples, 1)),-1)

  x = ray_origins.unsqueeze(1) + t.unsqueeze(2) * ray_directions.unsqueeze(1)
  ray_directions = ray_directions.expand(nb_bins, ray_directions.shape[0], 3).transpose(0,1)

  return x, ray_directions, delta


###
def render_rays(nerf_model, ray_origins, ray_directions, hn=0, hf=0.5, nb_bins=192, stage='coarse'):
  device = ray_origins.device
  nb_samples = ray_origins.shape[0]

  # course sampling
  x_coarse, r_d_coarse, delta_coarse = coarse_sample(ray_origins, ray_directions, hn, hf, nb_bins, nb_samples)
  x, r_d, delta = x_coarse, r_d_coarse, delta_coarse

  if stage == 'fine':
    # fine sampling
    c, sigma = nerf_model(x.reshape(-1,3), r_d.reshape(-1,3))
    sigma = sigma.reshape(nb_samples,nb_bins)

    x_fine, r_d_fine, delta_fine = fine_sample(sigma, ray_origins, ray_directions, hn, hf, nb_bins, nb_samples)

    # cancating sampling

    x = torch.cat((x_fine, x_coarse),-1)
    r_d = torch.cat((r_d_fine, r_d_coarse),-1)
    delta = torch.cat((delta_fine, delta_coarse),-1)

    x, r_d, delta = x_fine, r_d_fine,delta_fine

  #print("delta",delta)
  # final rendering
  colors, sigma = nerf_model(x.reshape(-1,3), r_d.reshape(-1,3))

  colors = colors.reshape(x.shape)
  sigma = sigma.reshape(x.shape[:-1])
  #print("sigma",sigma)

  alpha = 1 - torch.exp(-sigma * delta)

  #print("alpha:",alpha)
```

```
  weights = compute_accumulated_transmittance(1-alpha).unsqueeze(2) * alpha.unsqueeze(2)
  c = (weights * colors).sum(dim=1)
  weight_sum = weights.sum(-1).sum(-1)
  return c + 1 - weight_sum.unsqueeze(-1)
```

Sampling from image (samples.py):

```

import torch
import numpy as np
import os
import cv2 as cv

def resize_img(img,scale):
  scale_percent = scale # percent of original size
  width = int(img.shape[1] * scale_percent / 100)
  height = int(img.shape[0] * scale_percent / 100)
  dim = (width, height)

  # resize image
  resized = cv.resize(img, dim, interpolation = cv.INTER_AREA)
  return resized

def get_odg(img,u,v,E,hwf,z):
  #o = -E[:3,:3].matmul(E[:3,3])
  o = E[:3,3]
  #px_2_world(hwf[0]//2,hwf[1]//2, E, hwf, 0)
  #d = fp-o
  d = torch.zeros(3,1)
  d[0],d[1],d[2] = u-hwf[0]/2,v-hwf[1]/2,hwf[2]
  d = d.to(torch.float32)
  #d = E[:3,:3].transpose(0,1).matmul(d).reshape(-1)
  d = E[:3,:3].matmul(d).reshape(-1)

  #fp = px_2_world(u,v, E, hwf, z)
  d = -d/(torch.sqrt((d*d).sum(-1))+1e-10) + 1e-10

  g = img[u,v]/256
  return o,d,g

def sample_tt(basedir, nm_cams, to_test=5):
  #basedir = "/content/drive/MyDrive/github-project/My_Nerf/datasets/fufu"
  nm_testing = nm_cams//to_test
  nm_training = nm_cams - nm_testing

  poses_arr = np.load(os.path.join(basedir, 'poses_bounds.npy'))

  info = poses_arr[:, :-2].reshape([-1, 3, 5])#.transpose([1,2,0])
  #bds = poses_arr[:, -2:].transpose([1,0])
  z = 10 #int(torch.means(bds))

  imgs = [os.path.join(basedir, 'images', f) for f in sorted(os.listdir(os.path.join(
    basedir, 'images'))) \
        if f.endswith('JPG') or f.endswith('jpg') or f.endswith('png')]

  scale = 20
  hwf = torch.from_numpy(info[0,:3,-1])
  hwf[0], hwf[1] = hwf[0]*(scale/100), hwf[1]*(scale/100)
  print(int(hwf[0]))
  print(int(hwf[1]))


  poses = torch.from_numpy(info[:,:3,:4])

  Es = torch.zeros((poses.shape[0],1,4))
```

```
  Es[:,:,-1] = 1
  Es = torch.concat((poses, Es), axis = 1)

  ray_one_img = int(hwf[0]*hwf[1])

  nm_training *= ray_one_img
  nm_testing *= ray_one_img

  training = torch.ones([nm_training,9])
  testing = torch.ones([nm_testing,9])

  print(training.shape)
  print(testing.shape)

  off_training = 0
  off_testing = 0

  for i in range(nm_cams):
    print(i)

    E = Es[i].to(torch.float32)
    img = cv.imread(imgs[i])
    img = resize_img(img, scale)
    img = torch.from_numpy(cv.resize(img,(int(hwf[1]),int(hwf[0]))))

    off = 0
    for j in range(int(hwf[0])):
      for k in range(int(hwf[1])):
        u,v = j,k
        o,d,g = get_odg(img,u,v,E,hwf,z)

        if (i+1)%to_test != 0:
          # traning
          training[off_training*ray_one_img + off,:3] = o
          training[off_training*ray_one_img + off,3:6] = d
          training[off_training*ray_one_img + off,6:] = g

        else:
          # testing
          testing[off_testing*ray_one_img + off,:3] = o
          testing[off_testing*ray_one_img + off,3:6] = d
          testing[off_testing*ray_one_img + off,6:] = g

        off += 1
    if (i+1)%to_test != 0:
      off_training += 1
    else:
      off_testing += 1

  return training, testing
```

Visualise component/part (visualisation.py):

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.patches import Patch
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from pytorch3d.transforms import euler_angles_to_matrix

class Visualizer:
    def __init__(self, xlim, ylim, zlim):
        self.fig = plt.figure(figsize=(18, 7))
        #self.ax = self.fig.gca(projection='3d')
        self.ax = self.fig.add_subplot(projection='3d')
```

```
        self.ax.set_aspect('auto')
        self.ax.set_xlim(xlim)
        self.ax.set_ylim(ylim)
        self.ax.set_zlim(zlim)
        self.ax.set_xlabel('x')
        self.ax.set_ylabel('y')
        self.ax.set_zlabel('z')
        self.ax.view_init(45, 90)
        print('initialize camera pose visualizer')

    ## visulising camera

    def extrinsic2pyramid(self, extrinsic, color='r', focal_len_scaled=5, aspect_ratio
    =-0.3):
        vertex_std = np.array([[0, 0, 0, 1],
                               [focal_len_scaled * aspect_ratio, -focal_len_scaled *
    aspect_ratio, focal_len_scaled, 1],
                               [focal_len_scaled * aspect_ratio, focal_len_scaled *
    aspect_ratio, focal_len_scaled, 1],
                               [-focal_len_scaled * aspect_ratio, focal_len_scaled *
    aspect_ratio, focal_len_scaled, 1],
                               [-focal_len_scaled * aspect_ratio, -focal_len_scaled *
    aspect_ratio, focal_len_scaled, 1]])
        vertex_transformed = vertex_std @ extrinsic.T
        meshes = [[vertex_transformed[0, :-1], vertex_transformed[1][:-1],
    vertex_transformed[2, :-1]],
                  [vertex_transformed[0, :-1], vertex_transformed[2, :-1],
    vertex_transformed[3, :-1]],
                  [vertex_transformed[0, :-1], vertex_transformed[3, :-1],
    vertex_transformed[4, :-1]],
                  [vertex_transformed[0, :-1], vertex_transformed[4, :-1],
    vertex_transformed[1, :-1]],
                  [vertex_transformed[1, :-1], vertex_transformed[2, :-1],
    vertex_transformed[3, :-1], vertex_transformed[4, :-1]]]
        self.ax.add_collection3d(
            Poly3DCollection(meshes, facecolors=color, linewidths=0.3, edgecolors=color,
    alpha=0.35))

    def customize_legend(self, list_label):
        list_handle = []
        for idx, label in enumerate(list_label):
            color = plt.cm.rainbow(idx / len(list_label))
            patch = Patch(color=color, label=label)
            list_handle.append(patch)
        plt.legend(loc='right', bbox_to_anchor=(1.8, 0.5), handles=list_handle)

    def colorbar(self, max_frame_length):
        cmap = mpl.cm.rainbow
        norm = mpl.colors.Normalize(vmin=0, vmax=max_frame_length)
        self.fig.colorbar(mpl.cm.ScalarMappable(norm=norm, cmap=cmap), orientation='
    vertical', label='Frame Number')

    def show(self):
        plt.title('Extrinsic Parameters')
        plt.show()

    def get_color(h):
      hex_string = h[2:]
      hex_string = '#' + '0' * (6 - len(hex_string)) + hex_string
      return hex_string
```