PicaNeRF: Neural Radiance Fields with Cubic Stylization

Rikin Gurditta
rgurditt@uwaterloo.ca
University of Waterloo
Waterloo, Canada

Yuri Boykov
yboykov@uwaterloo.ca
University of Waterloo
Waterloo, Canada

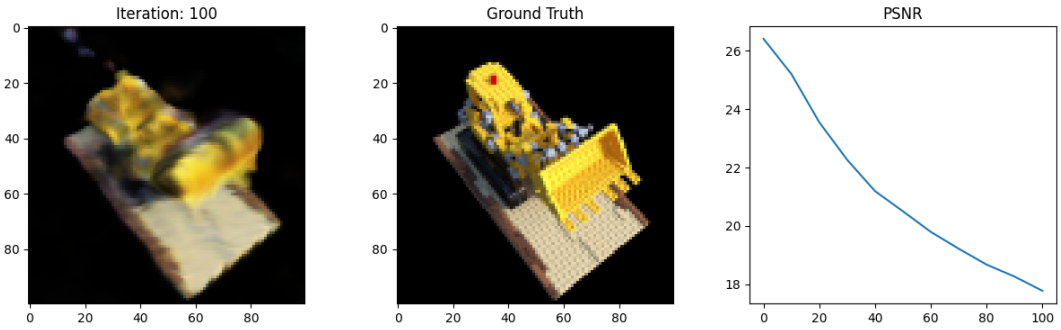


Figure 1: The useless result produced by our method. Crumpled and vaporized: A terrible fate.

Abstract

Neural radiance fields [6] (NeRF) provide a method of constructing 3D geometry representing a scene given a sparse set of input views. Cubic stylization [4] is a method by which an input shape can be given the style of a cube while retaining its geometric features. Our method encourages cubic stylization during the optimization of a NeRF by introducing a gradient-based regularization.

CCS Concepts

• **Computing methodologies** → **Reconstruction**; **Shape modeling**.

Keywords

Neural Radiance Fields, Reconstruction, Stylization, Regularization

ACM Reference Format:
Rikin Gurditta and Yuri Boykov. 2018. PicaNeRF: Neural Radiance Fields with Cubic Stylization. In *Proceedings of CS 898 Winter 2024 (CS 898 '24).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Neural radiance fields (NeRF) presented a novel and exciting method of novel view synthesis given a sparse set of views of the scene [6]. New views are output using a standard volume rendering algorithm, which integrates a *radiance field* F_Θ of scene density and colour. As a corollary to their method of novel view synthesis, Mildenhall et al. introduced the use of F_Θ as a versatile implicit representation of 3D geometry. This expanded our "vocabulary" for expressing meshes beyond existing representations such as polygonal/volumetric meshes, voxel grids, point clouds, and signed distance fields (SDFs). Due to its novelty, geometry processing tasks that are well-studied on previous representations must be reformulated for NeRF.

Cubic stylization is a method by which an input shape can be deformed into the style of a cube while retaining its original content [4]. In other words, the shape remains the same semantically, but its surfaces are axis-aligned. Shape modeling and deformation methods allow users to perform sophisticated operations on 3D meshes with less manual labour and more consistency in the quality of their meshes. These tools are mature for explicit shape representations such as 3D meshes, which are ubiquitous in 3D art and entertainment, and CAD, which is ubiquitous in engineering. In this paper we aim to contribute a shape modeling method that can be applied to a NeRF, allowing for artistic control and deformation of learned geometries.

We propose PicaNeRF, a method of encouraging cubic stylization during the optimization of a neural radiance field. We introduce a regularization loss similar to the cubeness term presented by Liu and Jacobson [4]. Due to our representation of geometry using a NeRF, our method can apply stylization not only to input 3D meshes, but also to real-life scenes learned from a sparse set of input photographs.

This paper will present my findings in attempting to implement PicaNeRF in a traditional SIGGRAPH conference proceedings format. However its purpose is to demonstrate my work done this semester for a course project, so some parts may not read the same way as a conference or journal paper.

CS 898 '24, Jan 01–Aug 31, 2024, Waterloo, ON

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

2 Related works

2.1 Shape deformation and stylization

As-rigid-as-possible deformations are constructed by assigning a transformation R_i to each vertex i of a mesh, and optimizing it to encourage rigidity:

$$\underset{\{\tilde{\mathbf{v}}_i\},\{R_i\}}{\text{minimize}} \sum_i w_i \sum_{j \text{ adj to } i} \frac{w_{ij}}{2} \left\| R_i(\mathbf{v}_j - \mathbf{v}_i) - (\tilde{\mathbf{v}}_j - \tilde{\mathbf{v}}_i) \right\|^2$$

where $\{\mathbf{v}_i\}$ and $\{\tilde{\mathbf{v}}_i\}$ are respectively the rest and deformed vertex positions, w_i, w_{ij} are weights related to the original geometry [9]. Note that the local transformations $\{R_i\}$ and the vertex positions $\{\mathbf{v}_i\}$ are jointly optimized (using a local-global ADMM scheme). The elegance of this approach relies on the explicit representation of the geometry - it can be directly deformed by modifying the positions of the vertices making up the mesh. ARAP aims to minimize local deformation, so it is reasonable

Cubic stylization [4] takes as input a 3D mesh with rest vertices $\{\mathbf{v}_i\}$. For each vertex i it determines a deformed position $\tilde{\mathbf{v}}_i$ and a local rotation $R_i \in SO(3)$. ADMM, a local-global optimization scheme, is used to jointly minimize the following energy with respect to $\tilde{\mathbf{v}}_i$ and R_i:

$$\min_{\{\tilde{\mathbf{v}}_i\},\{R_i\}} \sum_i \sum_{j \text{ adj to } i} \frac{w_{ij}}{2} \left\| R_i(\mathbf{v}_j - \mathbf{v}_i) - (\tilde{\mathbf{v}}_j - \tilde{\mathbf{v}}_i) \right\|_2^2 + \lambda a_i \left\| R_i \hat{\mathbf{n}}_i \right\|_1$$

λ is a parameter that determines the emphasis on cubeness, $\hat{\mathbf{n}}_i$ is the unit normal to the surface at vertex i, and w_{ij}, a_i are fixed weights relating to the original geometry of the mesh. The first term is the ARAP energy - optimizing this aims to make the transformation seem rigid when localized to each vertex. The second term is the cubeness - by minimizing the L^1 norm of the unit surface normal, we induce sparsity in it, which results in axis-alignment of the surface.

This formulation relies on the input being a 3D mesh, an explicit representation of the geometry. In particular, it relies on weighing the importance w_{ij} of an edge (i, j) when optimizing deformation for rigidity, and weighing the importance a_i of a vertex i's normal when determining cubeness. These allow for less prominent features to conform less to our goals of minimizing local deformations and axis-aligning the normals. This is key to the algorithm: in the extreme case, if every edge had to be rigidly transformed, then the entire mesh would be rigidly transformed and thus not deformed at all. If every normal had to be axis-aligned, certain geometries would be impossible to optimize (e.g. a tetrahedron). Working with implicit geometry, there are no canonical analogs to these quantities.

2.2 Neural implicit geometry

Methods of representing geometry generally fit into one of two classes: explicit and implicit. For our purposes we will stick to surface geometry embedded in 3D.

Explicit representations parameterize the surface of the geometry, often in a piecewise fashion [1]. The most common explicit representation in computer graphics is the *polygon mesh*, which consists of a set of vertices and a set of polygonal faces made up by subsets of vertices. This is convenient for geometry processing as it allows for manipulations in the domain of the manifold's intrinsic dimension rather than the embedded dimension, e.g. a sphere embedded in $\mathbb{R}^3$ can be processed using its 2D parameterization. Furthermore, this is convenient as it is immediately known "where" in space the surface is located.

By contrast, the geometry of implicit surfaces is described by a function labelling each point in space with information about the geometry [1] (we diverge from the definition provided by Botsch et al. [1] to provide a more general characterization). For example, a *signed distance field* is a function $f : \mathbb{R}^3 \to \mathbb{R}$ where $|f(\mathbf{x})|$ is the distance from $\mathbf{x}$ to the closest point on the surface and $\text{sgn}(f(\mathbf{x}))$ indicates whether $\mathbf{x}$ is inside or outside the (orientable) surface. With this representation it may be difficult to directly deform a surface with user-defined modifications, as local changes may have global consequences. It may also be more difficult to carry out common operations such as intersection tests, since the lack of explicit parameterization means that analytic solutions may not exist in general. Instead, root finding or search algorithms may be required to locate the surface.

Relatively recent in the field of computer graphics are implicit geometry representations using neural networks. The expressivity and straightforward training processes of neural networks make them a good choice for modeling such complex functions as implicit shapes.

One approach is to train a neural network to encode a signed distance field, allowing the exact value of the SDF to be evaluated at any point without employing a spatial discretization relying on samples and interpolation. Some work has been done on geometry processing with neural SDFs, such as constructive solid geometry operations [5] and levels-of-detail [10].

Another approach is neural radiance fields [6], which assign to each point a radiance $\sigma > 0$ whose magnitude indicates whether the point is part of the object. This representation is unlike others in that it cannot exactly represent a 2D manifold embedded in 3D space, rather it encodes a continuously varying "volumetric density" indicating how significantly a point contributes to a view of the shape.

2.3 Regularization of neural geometry

The RegNeRF method proposes regularizing neural radiance field optimization by rendering patches from novel camera angles and assessing them using a normalizing flow model Niemeyer et al. [7]. In doing this, one of the regularizations employed is a depth smoothness term:

$$\mathcal{L}_{DS} = \sum_{i,j} \left(\hat{d}_\theta(\mathbf{r}_{i,j}) - \hat{d}_\theta(\mathbf{r}_{i+1,j}) \right)^2 + \left(\hat{d}_\theta(\mathbf{r}_{i,j}) - \hat{d}_\theta(\mathbf{r}_{i,j+1}) \right)^2$$

for rays $\{\mathbf{r}_{i,j}\}$ making up a "patch". Formulating smoothness in this way means that an estimation of depth is required, even though a NeRF only encodes volumetric information. They formulate depth as

$$\hat{d}_\theta(\mathbf{r}) = \int_{t_{\min}}^{t_{\max}} T(t)\sigma(\mathbf{r}(t))t \, \mathrm{d}t$$

This can be interpreted as a weighted average depth along the ray, with weights of transmittance and density (see 3.1).

The same estimation of depth is applied in DiffNerf [2], however smoothness is formulated as the analytic gradient of this depth

(clipped at $g_{\max}$ to allow for sharp edges):

$$\mathcal{L}_{\text{depth}} = \sum_{i,j} \text{clip}\left(\nabla_{\mathbf{x}} \hat{d}_\theta(\mathbf{r}_{ij}), g_{\max}\right)$$

The analytic gradient can be required because the method replaces the activations in the NeRF architecture with their smooth analogues, allowing them to be computed using automatic differentiation. DiffNeRF also proposes regularization with the higher order quantities mean curvature and Gaussian curvature, which are also computed using automatic differentiation.

Gropp et al. [3] present a method of learning a neural signed distance field given a point cloud $\mathcal{X}$. An SDF ϕ has a value of 0 at points on the surface, so the method attempts to fit a function ϕ_θ whose 0-isosurface contains $\mathcal{X}$, i.e. it encourages that $\phi_\theta(\mathbf{x}) = 0$ for $\mathbf{x} \in \mathcal{X}$. Furthermore, it regularizes based on the *Eikonal* property of SDFs - $\nabla_{\mathbf{x}}\phi(\mathbf{x}) = 1$ wherever this gradient exists. This is because the $\phi(\mathbf{x})$ is the shortest distance from $\mathbf{x}$ to a point on the surface, so the gradient should point in the direction of maximum increase, i.e. the direction exactly away from the surface. Like DiffNeRF, this method also relies on calculating $\nabla_{\mathbf{x}}\phi_\theta$ using automatic differentiation. The Eikonal regularization is incorporated by adding a new term to the loss:

$$\mathcal{L} = \mathcal{L}_{\mathcal{X}} + \lambda \mathbb{E}\left[\nabla_{\mathbf{x}}\phi_\theta(\mathbf{x}) - 1\right]^2$$

The method is formulated for the expectation $\mathbb{E}$ calculated over an arbitrary distribution.

Our proposed method similarly regularizes using a function of the gradient with respect to the input $\nabla_{\mathbf{x}}\sigma$ calculated using automatic differentiation. Unlike RegNeRF and DiffNeRF, we do not only calculate gradients at the surface. Our approach is more similar to the method of Gropp et al. [3] - we calculate the expected value of the gradient over a volumetric domain. However, ours is not formulated for arbitrary distributions, only for the entire domain.

- DiffNerf [2] - also tries to modify surface normal. does it differently, by using normals from viewpoint. maybe this is better because it only deals with surface normals
- Implicit Regularization [3] wants to enforce eikonal property, so tries to enforce $\|\nabla_{\mathbf{x}} f(\mathbf{x})\| = 1$. this is evidence that dealing with $\nabla_{\mathbf{x}} f$ actually works.

3 Method

3.1 Neural radiance field

A radiance field is a function $F : (\mathbf{x}, \mathbf{d}) \mapsto (\sigma, \mathbf{c})$. It assigns a density σ each position $\mathbf{x}$ in our scene Ω. Given $\mathbf{x}$ as well as a viewing direction $\mathbf{d}$ it also assigns a colour $\mathbf{c}$ of the object from that direction. The independence of σ from the viewing direction $\mathbf{d}$ encourages consistency between different views. As a result, the σ field can encode a 3D geometry of the scene, as opposed to a "non-physical" internal representation that may result in better renders.

A neural radiance field F_Θ is represented by a multi-layer perceptron with parameters Θ. We use the nerf-pytorch pypi package [11] as our base implementation due to difficulties with our own implementation. The $\mathbf{x} = (x, y, z)$ coordinates are given as input to the neural network, and connected to 4 fully connected layers with 128 channels each and ReLU activation, which output a 128 channel feature vector. This feature vector and the position are fed into another 4 fully connected layers of the same size, but the output is a 128 channel feature vector and the density σ at the point. This feature vector as well as the viewing direction $\mathbf{d}$ is fed into a last layer, which uses sigmoid activation to output the RGB colour values in $[0, 1]^3$. Note that this architecture guarantees that the density σ does not depend on the viewing direction $\mathbf{d}$.

We use the same positional encoding as presented in [6].

A new view of the scene is synthesized using a standard volume rendering approach. The light transport equation for a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ is

$$C(\mathbf{r}) = \int_{t_{\min}}^{t_{\max}} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})\, \mathrm{d}t$$

where

$$T(t) = \exp\left(-\int_{t_{\min}}^{t} \sigma(\mathbf{r}(t))\, \mathrm{d}s\right)$$

C is an integral of the colours $\mathbf{c}$ along the ray weighted by the density σ and the transmittance T. This latter factor is a quantity that decreases as more density has been "seen" along the ray, modeling opacity.

Given a camera angle θ, viewing directions for each pixel (i, j) the camera are computed, and a ray is cast in each of these directions to get a colour $C(\mathbf{r}_{ij})$. The pixel colours together create an image $I_\Theta(\theta) \in \mathbb{R}^{\text{width}\times\text{height}\times 3}$.

The model is trained using differential rendering. The training set consists of pairs $\{(\theta_i, I_i)\}_{i=1}^{M}$ of camera angles and ground truth images of the scene from those angles. Given a training point, the scene is rendered from its angle and the loss is computed as mean square error using the standard L^2 norm:

$$\mathcal{L}_{\text{NeRF}} = \|I_\Theta(\theta_i) - I_i\|_2^2$$

3.2 Cubic stylization

Liu and Jacobson [4] implement cubic stylization by adding a regularization term to the ARAP energy to optimize:

$$\lambda a_i \left\|R_i \hat{\mathbf{n}}_i\right\|_1$$

$R_i\hat{\mathbf{n}}_i$ is the unit normal to the deformed surface, and the L^1 norm penalty encourages sparsity. We reformulate an analogous "cubeness" term for NeRF by first finding an analog to the surface normal.

The density component σ of the radiance field is a continuous function of the input position $\mathbf{x}$ within the scene. In a converged, well-resolved neural radiance field, only regions containing geometry would have high values of density. Using the mean value theorem, we expect the gradient of density to point in the direction perpendicular to the surface - it would be small along the surface and large moving away from it. We also expect that broad regions of low density may have small density gradients when the NeRF is well-resolved. Thus, the direction of $\nabla_{\mathbf{x}}\sigma$ may be a suitable approximation to the surface normal at each point, and we can calculate global cubeness by integrating over the entire domain.

Thus, we encourage cubeness by introducing a (λ-weighted) regularization term:

$$\mathcal{L}_{Pica} = \mathcal{L}_{\text{NeRF}} + \lambda \underbrace{\left\|\int_\Omega \hat{\nabla}_{\mathbf{x}}\sigma(\mathbf{x})\, \mathrm{d}\mathbf{x}\right\|_1}_{\mathcal{L}_{\mathbf{n}}}$$

We use $\hat{\nabla}$ to denote the normalized gradient, i.e.

$$\hat{\nabla} f = \begin{cases} \frac{\nabla f}{\|\nabla f\|} & \nabla f \neq \mathbf{0} \\ \mathbf{0} & \nabla f = \mathbf{0} \end{cases}$$

3.3 Discretization and numerics

The rendering integral is discretized as a summation over the ray. We choose a hyperparameter N to be the number of samples per ray. We divide the ray into N segments and sample uniformly from each of them. Let $L = t_{\min} - t_{\max}$ be domain of the parameterization of the ray. Then we choose sample points by sampling

$$t_i \sim \mathcal{U}\left[t_{\min} + (i-1)\frac{L}{N}, +t_{\min} + i\frac{L}{N}\right]$$

Let $\delta_i = t_{i+1} = t_i$ be the progress along the ray between sample points i and $i+1$, and let $\mathbf{x}_i = \mathbf{r}(t_i)$. Given these sample points, we numerically approximate our integral:

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma(\mathbf{x}_i)\delta_i))\mathbf{c}(\mathbf{x}_i)$$

where

$$T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma(\mathbf{x}_i)\delta_j\right)$$

The cubeness integral is also approximated using sampling. We choose a hyperparameter G for the grid sampling resolution. We divide the domain $\Omega = \left[-\frac{W}{2}, \frac{W}{2}\right]^3 \subset \mathbb{R}^3$ into a $G \times G \times G$ grid and sample uniformly from each cell. Define the interval $C_\ell = \left[-\frac{W}{2} + (\ell-1)\frac{W}{G}, -\frac{W}{2} + \ell\frac{W}{G}\right]$ to be the ℓ^{th} segment of the interval $\left[-\frac{W}{2}, \frac{W}{2}\right]$. Then we choose samples:

$$\mathbf{p}_{i,j,k} \sim \mathcal{U}(C_i \times C_j \times C_k)$$

We approximate the surface normal term of $\mathcal{L}_{\text{Pica}}$ by

$$\mathcal{L}_{\mathbf{n}} = \sum_{i,j,k} \hat{\nabla}_{\mathbf{x}} \sigma(\mathbf{p}) \left(\frac{W}{G}\right)^3$$

$\nabla_{\mathbf{x}}\sigma$ is computed analytically using Pytorch [8]. To reduce numerical instability, we threshold $\|\nabla_{\mathbf{x}}\sigma\|$ by a hyperparameter s before normalizing:

$$\hat{\nabla}\sigma = \begin{cases} \frac{\nabla\sigma}{\|\nabla\sigma\|} & \|\nabla\sigma\| > s \\ \mathbf{0} & \text{otherwise} \end{cases}$$

3.4 Training

First a vanilla NeRF was trained, i.e. a neural radiance field was optimized using only the original NeRF image loss $\mathcal{L}_{\text{NeRF}}$. Then, the network was trained further using our PicaNeRF loss $\mathcal{L}_{\text{Pica}}$. Training was divided into two stages as such due to the competing nature of the optimzation targets. $\mathcal{L}_{\text{NeRF}}$ encourages geometry to be accurate to the photographed object by penalizing inaccurate renders of the scene, however $\mathcal{L}_{\mathbf{n}}$ encourages inaccurate geometry by penalizing non-cubeness. What is desired is a deformation of an accurate object into a cubic-stylized one - in some sense, a projection of the fully-resolved NeRF onto the subspace of cubic-stylized NeRFs. We do not desire that cubeness be emphasized at every step of the optimization, as this does not help to obtain the accurate pre-projection geometry. This is analogous to the algorithm of Liu and Jacobson [4] in that its input is accurate geometry and its output is stylized. In our case, we must also discover our geometry, so we do so agnostic of the deformation step.

3.5 Software development

All code was written in Python using PyTorch [8]. Our Jupyter notebooks are included in our supplementary materials.

We first developed a 2D implementation of vanilla NeRF in order to achieve a minimal

In order to implement PicaNeRF, we began with the NeRF implementation provided on the Python Package Index by Trabucco [11]. The only necessary amendment was to implement the calculation of $\mathcal{L}_{\text{Pica}}$:

```
loss = ((pixels - batch['pixels']) ** 2).mean()
if pica:
    density, _ = nerf.forward(grid, torch.zeros_like(grid))
    (J, ) = torch.autograd.grad(density.sum(), grid, create_graph=True)
    k = J.norm(p=1).mean()
    J[J.norm(p=1) < k, :] = 0
    loss = loss + weight * normalize(J).norm(p=1).mean() * (W / G)^3
loss.backward()
```

4 Results and evaluation

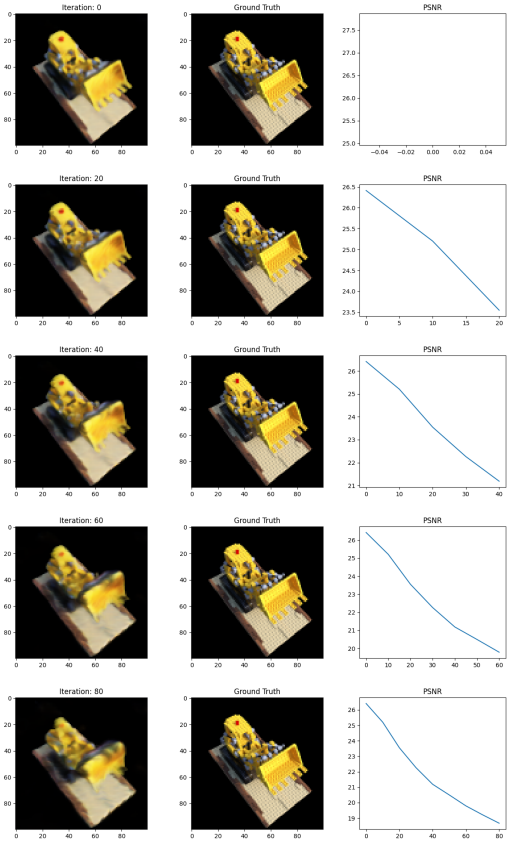
Using a grid size of $G = 32$ to sample the gradients, we could not produce any impressive results from our method. Larger grid sizes required more memory than was available on our Apple MacBook Air M2. Optimizing using the cubeness regularization term resulted in the geometry being "crumpled" and eventually "vaporized": It is clear that even the parts of the geometry that were approximately flat before did not end up being very flat.

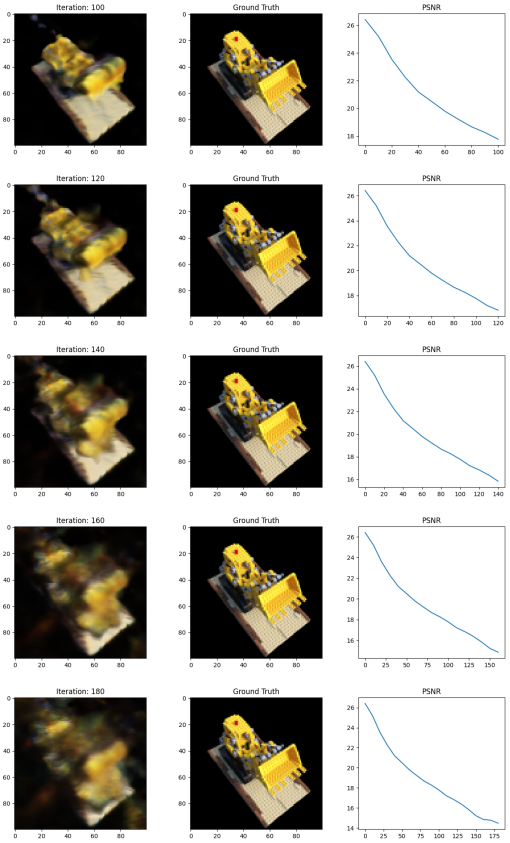
5 Further work

The proposed method seems to have broadly failed, however has not been fully investigated. It is possible that modifications in its formulation and implementation without a radical departure from its approach could yield better results. First, there are a number of hyperparameters that should be explored more carefully: training rate α, samples per ray N, gradient threshold s, gradient integration grid size G, regularization weight λ.

These aspects can also be reformulated to better suit their purposes. For example, Gropp et al. [3] formulate their Eikonal regularization as the expected value of an arbitrary distribution over the domain. This could be a helpful approach when calculating our cubeness term as opposed to integrating over the whole domain, as it could allow us to focus our computation on occupied regions and sample them more densely. This could be more effective than simply sampling a finer grid over the whole domain.

On the flipside, this could also be done using the approach taken in DiffNeRF [2] - the gradient may need only to be calculated on (some estimation of) the surface geometry, since its purpose is deformation of surface geometry. Taking inspiration from RegNeRF [7], this could also be applied to patches calculated from unknown views since our cubeness does not require any ground truth to compare to, just like the smoothness regularization in RegNeRF.





Another direction of inquiry is the training process. We argue that imposing the cubeness loss on an underresolved NeRF is undesirable, as our goal is to create deformed geometry rather than geometry that has weak relation to the input data. However, while a NeRF is underresolved, it may vary more "smoothly" from regions of low density to regions of high density, which may lend itself better to regularization based on gradients. A simpler approach to the same idea would be applying a blur to the NeRF before sampling its gradient. This method has more grounding in numerical methods - it is common to apply a Gaussian blur to a function before downsampling it in order to avoid aliasing from high frequency changes around a sampling point. However, this may not be so simple for a NeRF, as a blurred field is an integrated quantity which may not be easy to calculate. In fact, it may be as numerically intractable as our original approach. At the same time, the approach proposed above of finding a better distribution to integrate over could be employed here too.

Acknowledgments

To my cat Shadow.

References

[1] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Levy. 2010. *Polygon Mesh Processing*. Routledge.

[2] Thibaud Ehret, Roger Marí, and Gabriele Facciolo. 2022. Regularization of NeRFs using differential geometry. arXiv:2206.14938 [cs.CV] https://arxiv.org/abs/2206.14938

[3] Amos Gropp, Lior Yariv, Niv Haim, Matan Atzmon, and Yaron Lipman. 2020. Implicit Geometric Regularization for Learning Shapes. arXiv:2002.10099 [cs.LG] https://arxiv.org/abs/2002.10099

[4] Hsueh-Ti Derek Liu and Alec Jacobson. 2019. Cubic Stylization. *ACM Transactions on Graphics* (2019).

[5] Zoë Marschner, Silvia Sellán, Hsueh-Ti Derek Liu, and Alec Jacobson. 2024. Constructive Solid Geometry on Neural Signed Distance Fields. *SIGGRAPH Asia 2023 Conference Papers* (2024).

[6] 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 *ECCV*.

[7] Michael Niemeyer, Jonathan T. Barron, Ben Mildenhall, Mehdi S. M. Sajjadi, Andreas Geiger, and Noha Radwan. 2022. RegNeRF: Regularizing Neural Radiance Fields for View Synthesis from Sparse Inputs. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[8] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[9] Olga Sorkine and Marc Alexa. 2007. As-Rigid-As-Possible Surface Modeling. In *Proceedings of EUROGRAPHICS/ACM SIGGRAPH Symposium on Geometry Processing*. 109–116.

[10] Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. 2021. Neural Geometric Level of Detail: Real-time Rendering with Implicit 3D Shapes. (2021).

[11] Brandon Trabucco. 2021. NeRF. https://github.com/brandontrabucco/nerf.