Faster than NERFs – 3D Models from 2D Images

Aniket Rajnish
Indian Institute of Technology, Gandhinagar, India
http://makra.wtf
aniket.r@iitgn.ac.in

Progyan Das
Indian Institute of Technology, Gandhinagar, India
http://progyan.me
progyan.das@iitgn.ac.in

Abstract

Slow inference and training times have always been an issue with Neural Radiance Fields, and voxel representations, often used in these papers, lead to prohibitively large memory requirements and very long waiting times. While the accuracy of NERFs are very high, we may be willing to sacrifice representation accuracy in the favour of time. We note that game-development, in particular, suffers from a large latency from ideation to the actual creation of assets for deployment.

In addition, although faster implementations of NERFs, like Instant-NGP [4] and Plenoxels [7] have been famously published in the past year, we wanted to devise an end-to-end tool for going from an image to a quickly deployable 3D model with the least number of steps. Traditional ML techniques use voxel, mesh, or point-cloud based rendering techniques – these are volumetric, and often are bound by high time complexities (voxel-based rendering, for example, runs at $O(N^3)$ where N is the number of *voxels*, or 3-dimensional pixels, we are rendering). Instead, we use Signed Distance Functions, a (somewhat) more complicated but overall less algorithmically complex solution for rendering, that sacrifices some benefits of volumetric rendering for speed.

To make this possible, we wrote our own raymarching rendering engine on $C\#$ and $HLSL$ (short for High-Level Shader Language, a C-like language for use in Direct3D applications[3]), implemented it in Unity, and combined it with a Convolutional Neural Network trained on a custom data-set made on Blender. Our functioning assumption is that most complex shapes that we may want to approximate may be built with a set of 35 primitive shapes and an accompanying boolean expression combining a few operations (union, intersection, subtraction). This is the foundational principle of Constructive Solid Geometry[8], and we have found that it works with great effect in our project.

Our end-product, SDFNet, is a tool built by game-developers, for game-developers, that takes a simple 2D image, and generates a 3D, surface-rendered model that is immediately deployable for development.

1 Implementing the State of the Art

Before we started writing our own rendering engine, we wanted to see how Neural Radiance Fields are implemented in `python`. To that end, we implemented a truncated version of the classical NERF paper from Nvidia Research, based on `pytorch3D` documentation[1].

1.1 Neural Radiance Fields (NERFs) through raymarching

The input is a number of images of the target, from different angles and their corresponding cameras, and our network attempts to build a scalar field that allows us to generate views from any other angle. To fit the radiance field, we render it from the viewpoints of the target cameras, and we compare the results with the observed target images and target silhouettes.

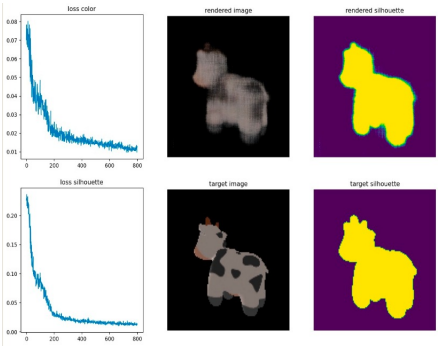


Figure 1 Side-by-side: the drop in the huber loss, and our trained model.

Loss is calculated as the mean huber loss (related to smooth-L1 loss) between rendered colours and the sampled target images, predicted masks and sampled target silhouettes.

We use two losses –

1. The color loss – as it is traditionally used in NERFs, the color loss essentially compares a snapshot of the model from the same camera position and angle with the data-point, which in this case is our corresponding image.

```
ray_color = sampled_images(
    target_images[batch_idx],
    sampled_rays.xys
)
color_error =  jnp.mean(
                jnp.abs(
                huber_loss(
    rendered_images,
    colors_at_rays,
)))
```

2. The silhouette loss – the silhouette loss forces the model to absorb the rays where necessary, and not pass through it, and vice-versa. This is possible because our dataset comes with segmentation-masks – otherwise, we could have used an architecture like Mask-RCNN for image segmentation and obtained a good approximate for the same.

```
silhouettes_at_rays =
                        sampled_images(
    target_silhouettes[batch_idx, None],
    sampled_rays.xys
)
sillhouette_err = jnp.mean(
                        jnp.abs(
                        huber(
    rendered_silhouettes,
    silhouettes_at_rays,
))
```

1.2 Drawbacks of NERFs

As we can see, current state-of-the-art techniques produce very accurate results. However, as the paper mentions, they are slow, and often prone to taking hours to train. While it is

possible to do away with neural networks and use classical machine learning for a speedup [7], we wanted to minimize the time-consuming process of training a radiance field altogether, and therefore, we decided to shift to neural networks only for *detecting* shapes and structures in our images, and not for *reconstructing* our models from the images.

2 Signed Distance Functions and Surface Rendering

Computationally, geometry is often stored explicitly as a list of points, triangles, or other geometric fragments [5]; however, these methods are computationally expensive, and we can devise both parametric and non-parametric methods for expressing these geometries implicitly. Signed Distance Functions, therefore, are a method for parametric implicit surface representation. [6]

These signed distance functions, or SDFs for short, are defined as continuous functions that, when passed the coordinates of a point in space, will return the shortest distance between that point and some arbitrary surface corresponding to that specific function. The sign of the return value indicates whether the point is inside that surface or outside (hence *signed* distance function).

For example, for a sphere centered at the origin, the standard SDF is mentioned below. [6]

$$f(p) = \vec{p} - \vec{r}$$

2.1 Rendering Shapes

We wrote an Image Effect shader to render objects directly in the screen space instead of creating instances of individual objects. We wrote a raymarching loop in the shader to render these shapes using their individual signed distance functions. All the parameters were taken from our model in a CSV file and were communicated to the shader from a *C#* script using Compute Buffers. The dimensional parameters were stored in a custom class of `Vector12` with 12 fields (maximum dimensional inputs that any shape can take) for floats, as the Shader language doesn't support dynamic arrays. So these parameters were communicated in the following way:

```
    dimensions[0] = new vector12(cyl.r, cyl.h, 0,0,0,0,0,0,0,0,0,0);
    dimensions[1] = new vector12(cap.r1, cap.r2, cap.h, 0, 0, 0, 0,
                                                    0, 0, 0, 0,0);
```

Computer buffers were also used to communicate other information like the number of shapes to be rendered and the blend factor for the operations for each shape. All the shapes are rendered on render texture in front of the camera, the dimensions of which are communicated to the shader.

2.2 Building a custom-editor in Unity

A custom editor was developed in Unity to aid the need to fine-tune the objects rendered on the screen The editor could be accessed using the Unity's inspector. The following parameters were governed using the Custom Editor –

Shape Operation	Color	Blend Factor	Dimension Factor

Similarly, following spatial parameters were governed by the Unity's inspector component –

Shape position	Shape Orientation (Quarternion/Euler Angles)

3 Predicting Shapes – designing the Pipeline.

We used a CNN architecture similar to *Alex Net* and built on top of that. The input to the model was an image of the object, in our example case, the bottle with a constructive geometry made of primary shapes. A total of 17 metadata entries, along with every image, were stored to reconstruct the model. These entries correspond to the presence, shape, and color of the sub-parts of the bottles. These were the output of the model.

3.1 Architecture and Parameters of Neural Network.

The architecture of the model consisted of CNNs, Pooling layers, activation functions such as ReLU, and dense linear layers towards the end. Finally, we received the 17 labels as output from the model, and these were read by the renderer. The output is received from the model in a `csv` sheet which includes –

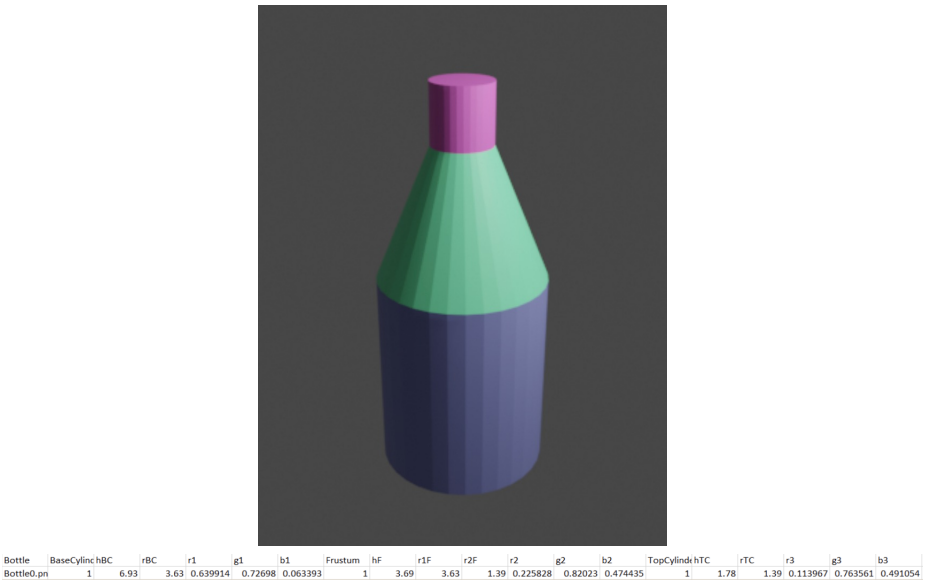
Parameter	Data-type	Description
Shape Index	`int`	Describes which out of a predefined list of primitives the given shape is.
Shape Position	`Vector3`	Describes the position of the shape in 3D space
Shape Rotation	`Vector3`	Describes the orientation of the shape in 3D space
RGB Values	`Vector3`	Denotes the RGB values of the color of the shape in the form of a 3-dimensional vector.
Shape Dimensions	`Vector12`	As a 12-dimensional vector, describes important absolute and relative dimensions for attributes like radius, height, et cetera, for the object. This is a sparse vector, and depending on the shape, many or none of the components may remain 0.

3.2 Creating our dataset in Blender

We used Blender to prepare the dataset of bottles of different shapes, sizes, and colours. Random gaussians were used to generate the dimensions of the bottles. The dimensions of the bottles along with the information of color were normalized before being fed to the model in the subsequent steps. The images and the information regarding the dimensions of the bottles were saved. Each bottle comprised two cylinders and one frustum. A total of 17 metadata entries, along with every image, were stored to reconstruct the model. These entries correspond to the presence, shape, and color of the sub-parts of the bottles. In the end, 400 distinct data points were generated and used.

3.3 In more detail: Parameters in use.

The *shape-index* is an int used to determine the shape we're trying to render. For instance, 2 denotes a cylinder, and 5 denotes a Frustrum. The *Shape Position* basically represents the represents predicted center of mass of the segmented shapes. The coordinates of which are scaled values of the coordinates of the pixel. Thus, these dimensions are not absolute



Bottle	BaseCylinc	hBC	rBC	r1	g1	b1	Frustum	hF	r1F	r2F	r2	g2	b2	TopCylinde	hTC	rTC	r3	g3	b3
Bottle0.pn	1	6.93	3.63	0.639914	0.72698	0.063393	1	3.69	3.63	1.39	0.225828	0.82023	0.474435	1	1.78	1.39	0.113967	0.763561	0.491054

Figure 2 A sample datapoint from the data-set, and the corresponding photograph.

but rather relative. The *Shape Rotation* values are the quaternion of the segmented shapes, again, it is calculated relative to the vertical axis. The *RGB* values are the average *RGB* values of each pixel on the segmented shapes. The *Shape Dimensions* are the individual dimensions required to define a particular shape. Take, for instance, a cylinder needs radius and height, a sphere just needs a radius, and a frustum/capped cone needs height, top, and bottom radius. Again, these are scaled values of the pixels covered.

3.4 Justification: Why *Unity*?

The Unity Engine was primarily used for the following reasons –

1. To aid in rendering the shadows using Unity's Directional Light Object. It takes its quaternion in Euler form (`Vector3`) into account.
2. To map the 2D image onto the screen space using Camera Frustrum (Matrix 4×4) and Camera to World Matrix (Matrix 4×4).

We wrote a custom Raymarcher class in *C#* to provide the following data manually –

Parameter	Data-type	Description
Shape Count	`int`	The length of the rows of the `csv` sheet, i.e, the number of shapes to be rendered on-screen. We later refactored this to be part of the output file.
Operation Index	`int`	An integer value used to denote which operation (union, intersection, subtraction) to perform with each shape.
Blend Factor	`float`, $0 \leq x \leq 1$	Whether or not to smoothen out the edges of different shapes, and finely blend them with each other.

Ideally, the model should have predicted these data-points as well, but we couldn't train it to do so at the moment and would be working on it further. We have found architectures like *CSG-Net*, that infer boolean expressions for Constructive Solid Geometry from 3D Models, that have piqued our interest, and we look forward to using them in our work.

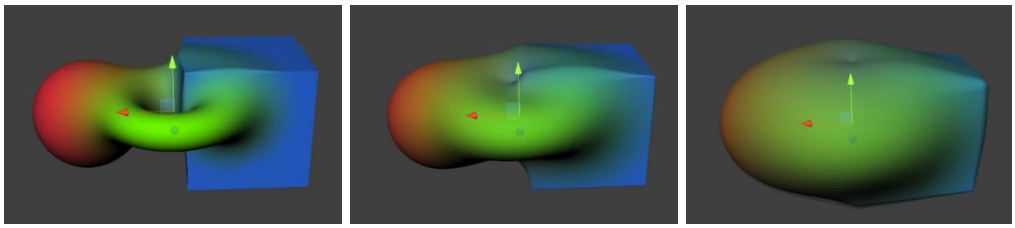


Figure 3 Left to right: How the blend factor increases from 0.36, to 0.59, to 0.93

3.5 Compute Buffer Conversion – *C#* to *HLSL*

All this data is passed to an HLSL-based shader to be rendered. This communication between the shader and C is done using a Compute Buffer of stride (size) 96 bytes.

Parameter	C#	HLSL (Compute Buffer Conversion)
Shape Index	`int`	`int`
Shape Position	`Vector3`	`float3`
Shape Rotation	`Vector3`	`float3`
RGB Values	`Vector3`	`float3`
Light Direction	`Vector3`	`float3`
Camera Frustrum	`Matrix` 4×4	`uniform float` 4×4
Camera to World Matrix	`Matrix` 4×4	`uniform float` 4×4
Shape Count	`int`	`int`
Operation Index	`int`	`int`
Blend factor	`float`	`float`

Apart from this the Image Effect shader used additional parameters –

1. Main Texture (`sampler2D`)
2. Camera Depth Texture (`sampler2D`)
3. Shapes (`Structured Buffer`)

The Main Texture and Camera Depth Texture is used to render multiple objects in the screen space without needing to create an instance for each. These are Image Effect shaders, that work like a post-processing effect over the screen-space, as opposed to vertex-shaders that work in world-space, which makes them more efficient and light for rendering crowded scenes. Please note that the entire SDF renderer is written in HLSL.

4 Putting it all together with raymarching.

We use raymarching for rendering – here, all attributes of the scene are implicitly defined in terms of some signed distance function. To find the intersection between the view ray and the scene, we start at the camera, and move a point along the view ray. At each step, we check if the SDF evaluates to a negative number at that point. If it does, we consider this a collision and initialize a surface at the point. [6][2]

This data is then used by the raymarching loop to decide the distance functions and operations for each shape and the parameters that these functions would use to render every

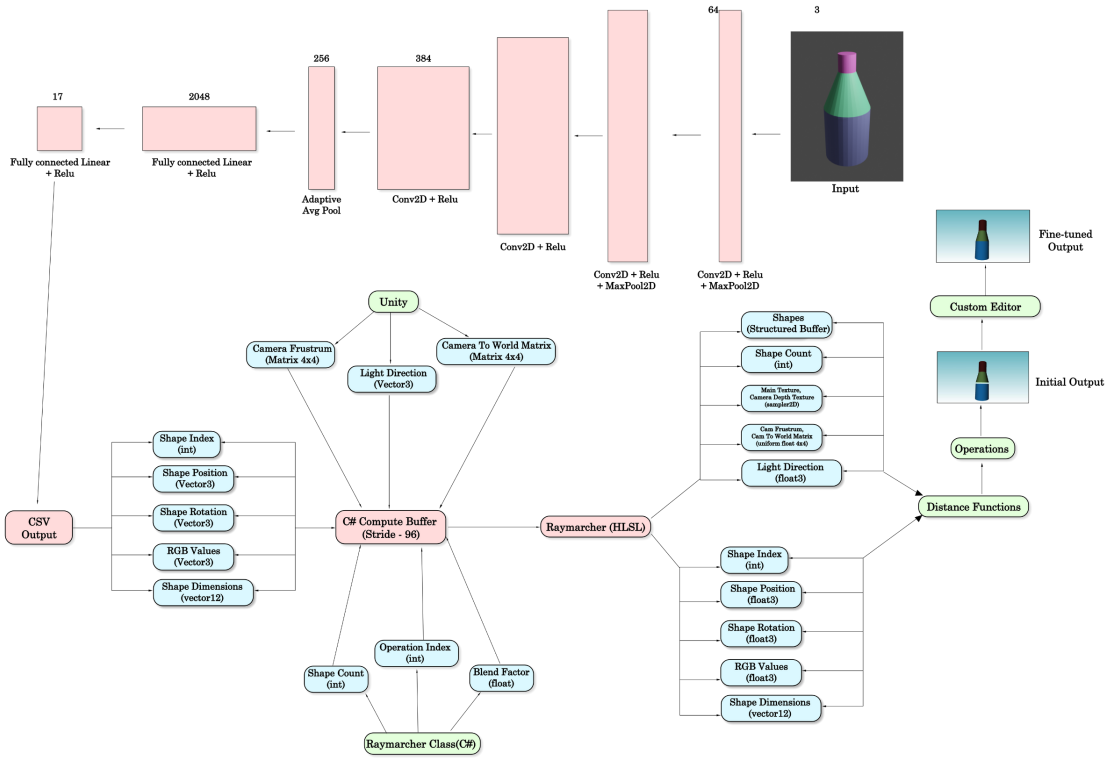


Figure 4 A flowchart of the entire pipeline for going from 2D image to 3D model

shape as perceived from the 2D image. The predicted model is surface-rendered and, as anticipated, comes with some flaws, which can later be fine-tuned using the custom editor that we wrote to reconstruct a fairly accurate model.

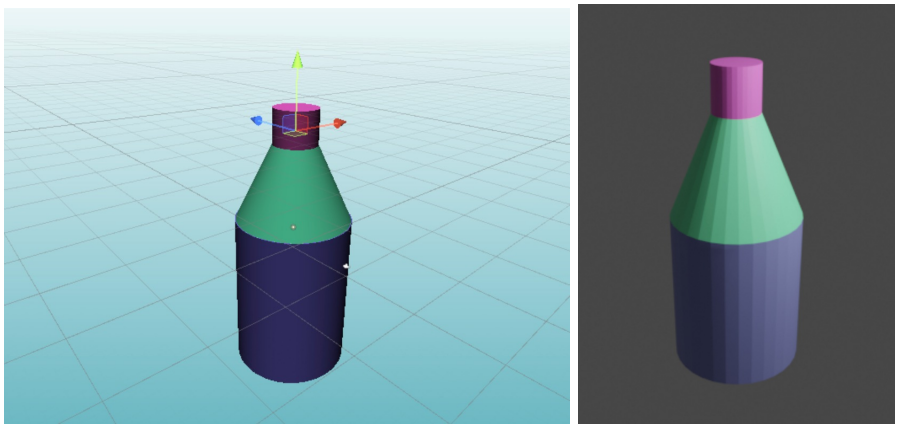


Figure 5 Side-by-side: reconstructed, SDF-rendered model, and the input image

5 Scope for improvement.

The pertinent areas of improvement for our tool have been listed below. These are areas of rapid development, and we foresee them being implemented very soon.

1. **Expanding for radially asymmetric geometries.**

   Since our tool only takes into account *one* image, we understand that for asymmetric geometries, there shall exist attributes that are occluded in any one camera angle. In addition, even for radially symmetric geometries, our model requires an angle that properly exposes all primitives present in the shape.

2. **Expanding for complex foregrounds and crowded backgrounds.**

   The model often fails for images with complex foreground geometries or crowded backgrounds. We wish to forego that limitation with a combination of foreground-background image segmentation, and some flavour of the *CSG-Net* model, to break images down to corresponding boolean expressions for constructive solid geometry.

3. **Raising number of primitives.**

   While we believe the 30 primitives that have been integrated into the Raymarching engine should be enough to express most geometries out there, there might be some esoteric, complex shapes that our model may not be able to approximate. We hope to soon get from 30 primitives to a planned 47 primitives.

4. **Predict Blend factor and Operations.**

   At the moment, our model does *not* predict the boolean operations and the blend factor. We wish that both can be implemented soon.

5. **Increase dataset variety.**

   Our model was only trained on a dataset consisting of frustrums and cylinders, out of the 30 primitives available. We wish to increase that variety with a much wider number of primitives.

6 Conclusion and acknowledgements

We have been able to produce a functioning prototype that can take a simple 2D image of a radially symmetric geometry and reconstruct a 3D representation through signed distance functions, with a combination of constructive solid geometry and neural networks. There is huge scope for improvement, and we are excited to keep working on this project, and also branch out into other domains.

We are thankful to Prof. Shanmuganathan Raman, for his guidance across the duration of the project. We are also grateful to his student, Ashish Tiwari, for his timely help in understanding hard concepts whenever we required it, and we are indebted to Shruhrid Banthia, a third-year undergraduate student at IIT Gandhinagar, who helped us build a significant portion of the tool.

References

1 Pytorch3d · a library for deep learning with 3d data. URL: `https://pytorch3d.org/tutorials/fit_simple_neural_radiance_field`.

2 Ray marching and signed distance functions. URL: `https://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/`.

3 Yong He, Kayvon Fatahalian, and Tim Foley. Slang: Language mechanisms for extensible real-time shading systems. *ACM Trans. Graph.*, 37(4), jul 2018. doi:10.1145/3197517.3201380.

4 Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, July 2022. doi:10.1145/3528223.3530127.

5 Jeong Joon Park, Peter Florence, Julian Straub, Richard A. Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. *CoRR*, abs/1901.05103, 2019. URL: http://arxiv.org/abs/1901.05103, arXiv:1901.05103.

6 Inigo Quilez. Signed distance functions. URL: https://iquilezles.org/articles/distfunctions/.

7 Sara Fridovich-Keil and Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In *CVPR*, 2022.

8 Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Csgnet: Neural shape parser for constructive solid geometry. *CoRR*, abs/1712.08290, 2017. URL: http://arxiv.org/abs/1712.08290, arXiv:1712.08290.