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.

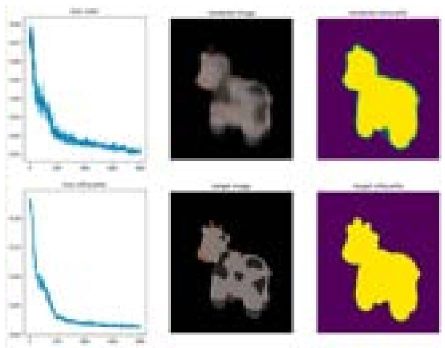


Figure 1 Side-by-side: the drop in the huber loss, and our trained model.

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

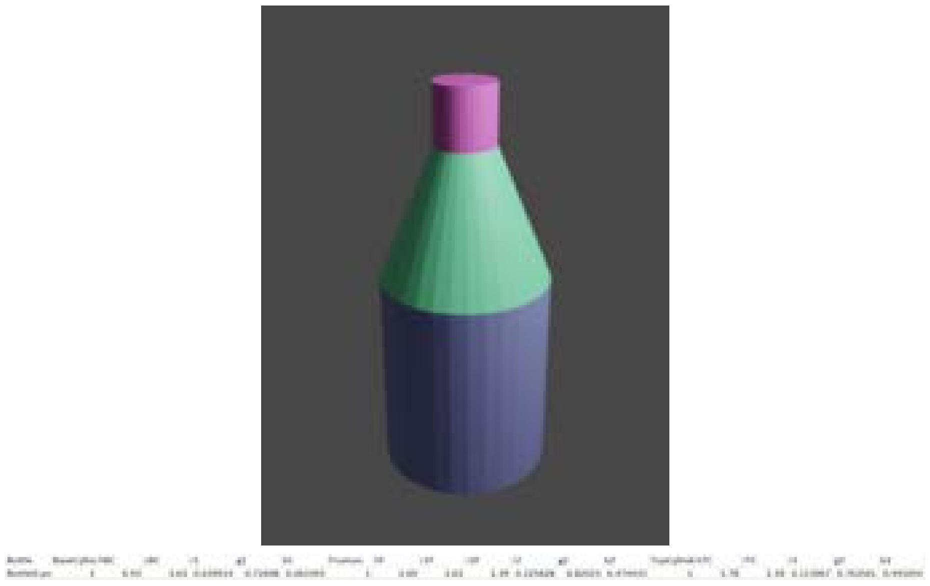


Figure 2 A sample datapoint from the data-set, and the corresponding photograph.

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.

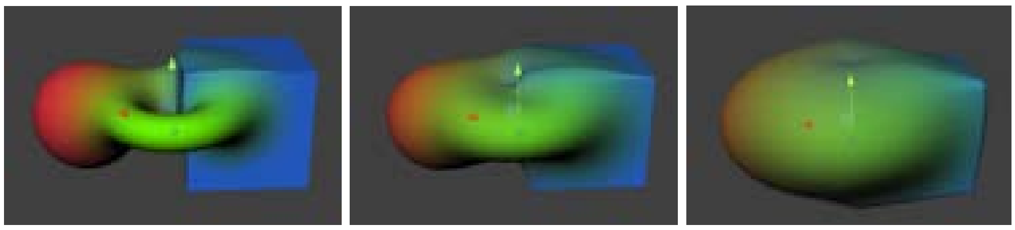


Figure 3 Left to right: How the blend factor increases from 0.36, to 0.59, to 0.93

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`

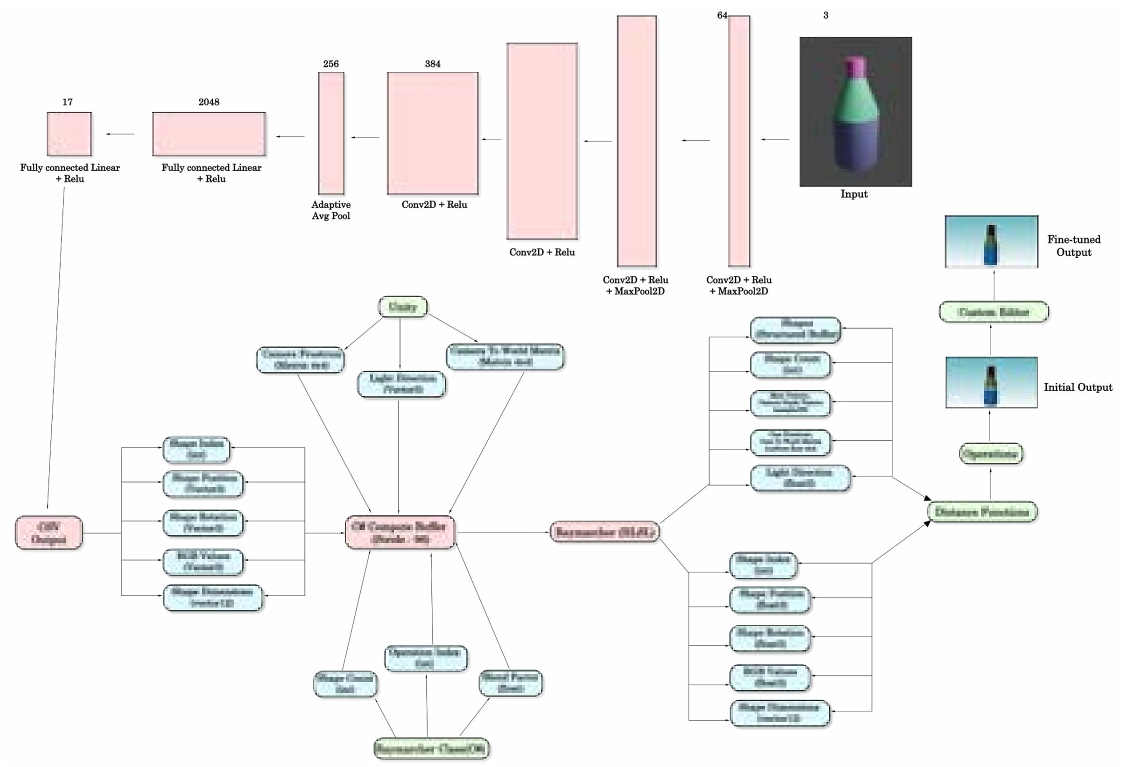
Apart from this the Image Effect shader used additional parameters –
1. Main Texture (`sampler2D`)
2. Camera Depth Texture (`sampler2D`)
3. Shapes (`Structured Buffer`)

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.

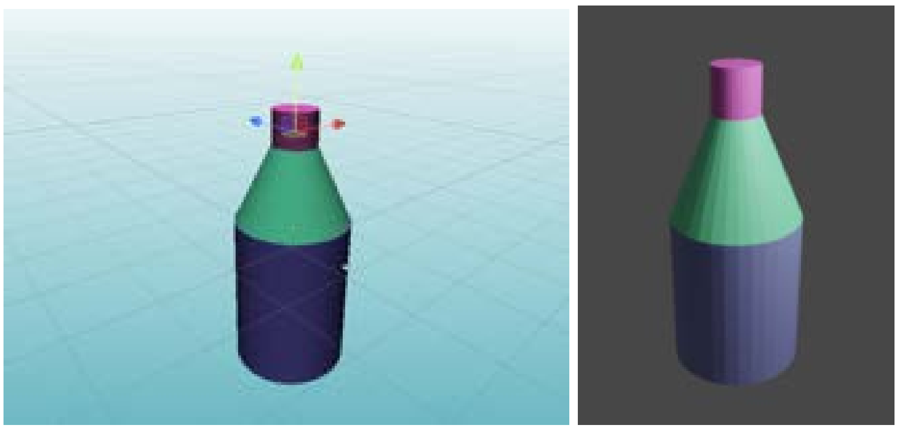
We use raymarching for rendering – here, all attributes of the scene are implicitly defined in terms of some signed distance function. To find the intersection between the view ray and the scene, we start at the camera, and move a point along the view ray. At each step, we check if the SDF evaluates 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.

Utilizing SDFNet to generate a comprehensive 3D shapes dataset and its Application

Aniket Rajnish
Indian Institute of Technology, Gandhinagar, India
`https://www.linkedin.com/in/makra2077/`
aniket.r@iitgn.ac.in

Progyan Das
Indian Institute of Technology, Gandhinagar, India
`https://www.linkedin.com/in/progyan20/`
progyan.das@iitgn.ac.in

Shruhrid Banthia
Indian Institute of Technology, Gandhinagar, India
`https://www.linkedin.com/in/shruhrid-banthia-b9aa01203/`
shruhrid.banthia@iitgn.ac.in

Abstract

*In this work, a comprehensive dataset of shapes generated using SDFNet, a proprietary rendering tool, is presented. The shapes in the dataset are created by manipulating primitive shapes using Boolean operations such as union, intersection, and subtraction. The dataset is intended to be used for Constructive Solid Geometry (CSG) algorithms and Disentangled Representation Learning. The dataset will be released along with a tool for generating personalized datasets. Furthermore, the work includes the use of FactorVAE*Kim and Mnih [2018]*, a popular disentanglement framework, to test the effectiveness of the generated dataset in training disentangled representations. The results of this study demonstrate that the dataset is capable of training FactorVAE to learn disentangled representations of the shapes in the dataset. We demonstrate the effectiveness of our approach by extracting 3D information from 2D images by traversing the latent space of FactorVAE. The proposed dataset and tool have the potential to disrupt the current leading VAE-based disentanglement frameworks.*

1 Introduction

This work presents a comprehensive dataset of low-poly 3D shapes generated using SDFNet, a proprietary rendering tool. The shapes in the dataset are created by manipulating primitive shapes using Boolean operations such as union, intersection, and subtraction. The dataset is intended to be used for Constructive Solid Geometry (CSG) algorithms and Disentangled Representation Learning. The dataset will be released along with a tool for generating personalized datasets.

The motivation behind this work is the need for a large publicly available dataset for low-poly 3D shapes and the challenge of generating accurate 3D models from 2D images. SDFNet was initially developed to create low-poly 3D models from a single 2D image. However, the tool required extensive shape-specific training, struggled to handle images with multiple shapes, and faced the challenge of the unavailability of a vast publicly available dataset for 3D shapes.

The availability of a diverse and large-scale 3D shape dataset plays a crucial role in extracting 3D information from 2D images. The proposed dataset with various primitive shapes, their dimensions, orientations, and color information can be useful in training VAEs to extract 3D information from 2D images. Recent works have shown that VAEs can be used effectively to disentangle and learn the underlying structure of 3D objects in their latent space. However, the complexity of generative factors can hinder the ability of VAEs to disentangle the learned latent factors.

The dataset is evaluated using FactorVAEKim and Mnih [2018], a popular disentanglement framework, to test the effectiveness of the generated dataset in training disentangled representations. The results demonstrate that the dataset is capable of training FactorVAE to learn disentangled representations of the shapes in the dataset. The proposed dataset and tool have the potential to disrupt the current leading VAE-based disentanglement frameworks.

2 Motivation

2.1 Constructive Solid Geometry

We had been developing a framework, SDFNet that aimed to create 3D low poly models from a single 2D image. Low-poly 3D shapes are simplified versions of high-poly 3D shapes, typically used in video games, virtual reality, and augmented reality applications. To make

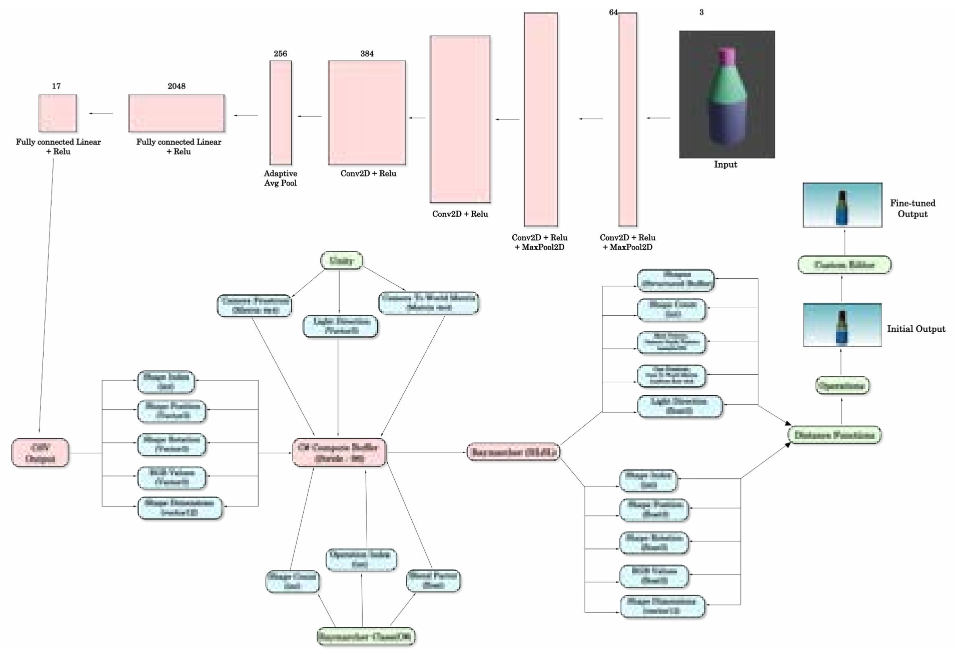


Figure 1 The Model architecture of SDFNet

the reconstruction of the models 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*He et al. [2018])*, implemented it in Unity, and combined it with a Convolutional Neural Network trained on a shape specific dataset made on Blender. However, during the implementation phase, we discovered some serious flaws that hindered its effectiveness.

- Firstly, the tool required extensive shape-specific training to accurately create the 3D model from a 2D image. This meant that we needed to train the model separately for each shape, which was a time-consuming process.

- Secondly, the tool could not render multiple shapes effectively. While it could generate 3D models from a single 2D image, it struggled to handle images with multiple shapes, resulting in inaccurate or incomplete reconstruction.
- Finally, and most importantly we faced the challenge of the unavailability of a vast publicly available dataset for 3D shapes. While there are many available 3D shape datasets, such as ShapeNet*Angel X. Chang*, ModelNet*Z. Wu and Xiao*, FAUST*Bogo et al. [2014]*, etc, there is a lack of publicly available datasets specifically focused on low-poly 3D shapes. The only dataset that aligns to our need was the 3DShapes*Burgess and Kim [2018]* dataset by Deepmind. This dataset was procedurally generated from 6 ground truth-independent latent factors, floor colour, wall colour, object colour, scale, shape and orientation. The dataset does not cover multiple shapes per image. Also, only 4 primitive shapes are covered in the dataset.

The availability of a large dataset of low-poly 3D shapes would be beneficial in several ways. It would allow for training models that can generate low-poly 3D shapes using techniques such as Constructive Solid Geometry. Such models could be used in various applications, including video game development and architectural design.

2.2 Disentangled Representation Learning

As stated in the pioneering paper with the same title *Wang et al. [2022]*, DRL "aims to learn a model capable of identifying and disentangling the underlying factors hidden in the observable data in representation form." Factor disentanglement is a field within machine learning that focuses on separating the underlying independent factors that contribute to generating complex data, such as images or 3D shapes. A definition of The goal is to identify and isolate these factors, such as shape, texture, color, orientation, and operation (in the case of CSG based data) so that they can be manipulated independently. It hypothesizes that the input data has been crafted using inherently independent factors of variation.

The availability of a diverse and large-scale 3D shape dataset plays a crucial role in extracting 3D information from 2D images. As pointed out in Neural 3D Mesh Renderer *Kato et al. [2018]*, learning 3D representations from 2D images requires large-scale datasets with a wide range of 3D structures. The proposed dataset with various primitive shapes, their dimensions, orientations, and color information, as mentioned later, can be useful in training VAEs to extract 3D information from 2D images.

Recent works have shown that VAEs can be used effectively to disentangle and learn the underlying structure of 3D objects in their latent space. However, the complexity of generative factors can hinder the ability of VAEs to disentangle the learned latent factors. The generative factors of 3D shapes are highly complex, and as a result, current disentangled VAEs may not fully capture the underlying factors of variation in the data unless provided with sufficient large and complex data for their training.

By providing a diverse and large-scale dataset with known ground truth information about the shapes, dimensions, orientations, and colors, the proposed dataset can help improve the ability of VAEs to disentangle the learned latent factors and extract 3D information from 2D images.

3 Dataset Description

3.1 Composition

- The individual parameters that govern each shape are–

- **Shape Index (int)**
- **Operation Index (int)**
- **Dimension Parameter (vector12)**
- **Position (Vector3)**
- **Orientation (Vector3)**
- **Color (HSV)**

- The **Shape Index** helps to identify the shape that has been used. It ranges from 0 to 30 while excluding the indexes for the fourteen shapes that are not considered for the datasets.
- The **Operation Index** helps to identify the operation associated with each shape. It ranges from (0,2).
- The **Dimension Parameters** help us to control the scale of the object. Given the datasets are generated using our propitiatory rendering engine, we can control individual dimensional parameters of objects like rounding factor, apex angle, thickness, etc., unlike just the X, Y, Z scale of the object, giving us more control over the shape's appearance.
- The **Position** of the shape can be any random position in a cube of dimension 2 units centered at the origin. The distance function of each shape is written such that its centroid lies on the position specified.
- The **Orientation** of the shape is randomly assigned to the shape using Random.rotation.
- The **Color** of the shape is given a random HSV value with hue between (0,1) and saturation and value fixed at 1.
- The dataset is completely randomized with each image in the dataset containing multiple shapes having varied parameters mentioned above.

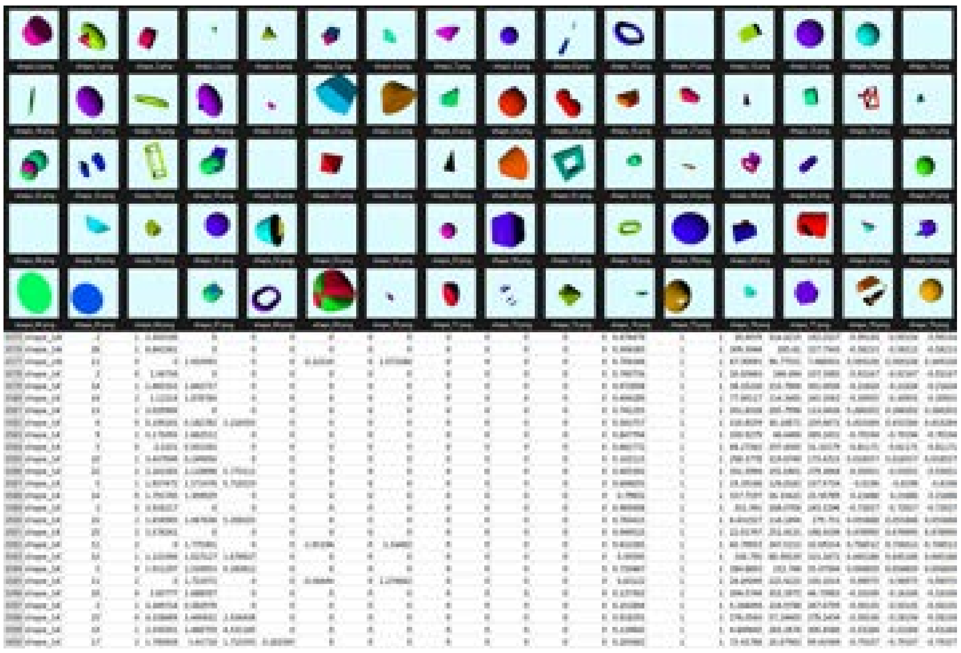


Figure 2 A glimpse of the Final Dataset.

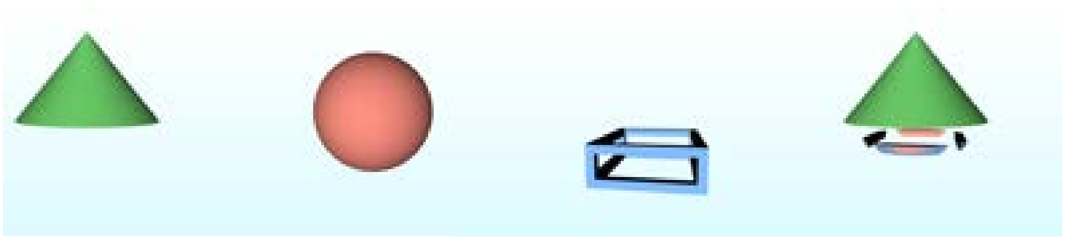


Figure 3 Sample image with operations: Cone (Union) + Sphere (Union) + Box Frame (Subtraction)

3.2 Preparation

We utilized Unity and the C# and HLSL programming languages to develop a custom raymarching engine for real-time rendering of complex shapes and scenes, in the form of an SDF Renderer. Raymarching is a technique for rendering three-dimensional scenes by tracing rays from the camera through the scene and computing the intersections with objects in the scene. Our engine used a distance function, defined in HLSL, to compute the distance from each ray to the nearest object in the scene, and a raymarching algorithm, implemented in C, to trace the ray through the scene and compute the final color of the pixel.

One of the key advantages of using Unity for this task was its built-in support for real-time rendering and shading. We were able to leverage Unity's powerful graphics pipeline and shader system to implement our raymarching engine. Additionally, Unity's flexible and customizable interface allowed us to easily experiment with different rendering techniques and optimizations.

The use of C and HLSL also provided several advantages for developing the raymarching engine. C is a modern, high-level programming language with a rich set of libraries and features, making it easy to implement complex algorithms and data structures. HLSL is a specialized programming language for defining shaders and rendering effects, with a syntax similar to C++. Together, these languages allowed us to create an efficient and scalable raymarching engine that could render complex scenes in real-time.

- The renderer supports over thirty primitives, three operations (Union, Intersection, and Subtraction), and RGB color values (along with shadows).
- The source code for the renderer can be found at `https://github.com/aniketrajnish/CS499-SDFNet/tree/main/Renderer`.
- The engine uses signed-distance functions (SDFs) to render individual shapes using the shape index, dimension data, and RGB values. Further, it knits them into the screen space using their relative position and dimensions.
- The tool is built on top of this Raymarching Engine and is deployed as an executable (.exe) file. It takes as input the following parameters-–
  - **Varying Camera Angle**
  - **Varying Orientation**
  - **Varying Position**
  - **Randomize Shape Count**
  - **Max Shape Count**
  - **Dataset Size**
  - **Dataset Path**
  - **Resolution**
  - **Shapes**

  - **Operations**
  - **Seed**
- The **Max Shape Count** int refers to the maximum number of shapes that each image in the Dataset should have.
- The **Randomize Shape Count** bool, if true, randomizes the number of shapes in each image between (0, n), where n is the "Max Shape Count". Else every image is generated with n number of shapes.
- The **Varying Camera Angle** bool assigns a different angle to the camera for each image of the dataset if true. Else the camera is just made to look at the object keeping it in the center using transform.LookAt().
- The **Variying Orientation** bool assigns a unique random orientation (angle) to the individual shapes constituting the dataset if true. Else the shapes are simply aligned with the axis using Quaternion.identity.
- The **Varying Position** bool assigns a unique random position to the individual shapes – in a cube of dimension 2 units centered at the origin – if true. Else the shapes are simply centered at the origin.
- The **Dateset Size** int refers to the number of images to be generated in the dataset.
- The **Dataset Path** string is used to determine the path where the dataset folder is to be created.
- **Resolution** takes the width and height of the images (in pixels) as input and generates them accordingly.
- **Shapes** and **Operations** are enums that determine which shape index and operation index are to be taken into consideration while generating each shape.
- The **Seed** int is used to input a seed value to generate a dataset that has already been created before by assigning the seed value to the Random State.

3.3 Processing And Labeling

Column Name	Info
filename	Name of the image file
shape	Shape Index
operation	Operation Index
a,b,c,d,e,f,g,h,i,j,k,l	dimensional parameters
hue, sat, val	HSV Values of the color
rot_x, rot_y, rot_z	Euler Angles
pos_x, pos_y, pos_z	Position Vector

Table 1 The label information of the dataset

- These parameters are exported in the CSV sheet with all the image information as shown in Figure 2
- The CSV sheet has the labeled columns as shown in Table1
- Each row depicts information about a shape in the image of a dataset.
- The seed value of each random state is also exported in a txt file and can be used to re-generate a dataset.

4 Experimentation

4.1 Brief outlook into Representation Learning and Disentanglement

4.1.1 Representation Learning

Representation learning has become a critical area of research in machine learning, particularly in the era of big data. The primary objective of representation learning is to learn a compact and informative representation of high-dimensional data that can be used for various downstream tasks such as classification, clustering, and retrieval. In recent years, representation learning has made significant progress, and its impact has been felt across different areas, including computer vision, natural language processing, speech recognition, and robotics.

4.1.2 Disentanglement

Disentanglement is a fundamental task in machine learning that involves separating the underlying factors of variation in high-dimensional data. The goal of disentanglement is to learn a set of independent and interpretable factors that capture the essential characteristics of the data. This task is particularly useful in situations where it is desirable to manipulate or control certain aspects of the data while preserving others. For example, in image generation, disentangled representations can be used to control specific attributes such as the shape, pose, and lighting of the generated image.

4.1.3 FactorVAE for disentanglement

FactorVAE is a state-of-the-art unsupervised learning technique for disentanglement that was introduced in 2018 by Kim et al. in their paper "Disentangling by Factorizing". The key idea behind FactorVAE is to encourage the learned representation to be invariant to certain latent factors of variation in the data, while still allowing the representation to capture the remaining factors of variation. This is achieved through a regularization term that encourages the learned representation to be minimally sensitive to small changes in the input data that are due to the disentangled factors of variation.

In particular, FactorVAE uses a variational autoencoder (VAE) to learn a probabilistic mapping from the input data to a low-dimensional latent representation. The VAE is regularized using an additional term called the total correlation penalty, which encourages the learned representation to be factorized by explicitly minimizing the dependence between the dimensions of the latent representation. This helps to ensure that each dimension of the representation captures a separate and meaningful factor of variation in the data.

Let x be the input data, and let z be the low-dimensional latent representation. The goal of FactorVAE is to learn a mapping from x to z that captures the underlying factors of variation in the data.

The mapping is parameterized by an encoder function $q(z|x)$ and a decoder function $p(x|z)$, which are learned through optimization of the following objective function:

$$\mathcal{L} = \mathbb{E}q(z|x)[\log p(x|z)] - \beta D\mathrm{KL}(q(z|x)||p(z))$$

where β is a hyperparameter that controls the strength of the regularization, and $D_{\mathrm{KL}}(q(z|x)||p(z))$ is the Kullback-Leibler (KL) divergence between the distribution of the latent representation under the encoder $q(z|x)$ and a prior distribution $p(z)$.

To encourage disentanglement of the learned representation, FactorVAE adds an additional penalty term to the objective function:

$$\mathcal{L}\mathrm{TC} = \mathbb{E}p(z)\left[\log q(z|x)\right] - \mathbb{E}_{q(z|x)}\left[\log q(z)\right],$$

where $q(z)$ is the marginal distribution of the latent representation under the encoder $q(z|x)$.

The final objective function used in FactorVAE is the sum of the reconstruction loss and the regularization term:

$$\mathcal{L}_{\text{final}} = \mathcal{L} + \lambda\mathcal{L}_{\text{TC}},$$

where λ is a hyperparameter that controls the strength of the total correlation penalty.

By minimizing the objective function $\mathcal{L}_{\text{final}}$, FactorVAE learns a disentangled representation that separates the underlying factors of variation in the data into independent and interpretable dimensions.

4.2 Iterations of Experiments to improve the dataset

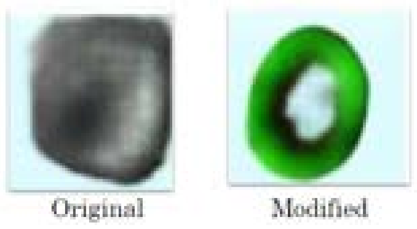


Figure 4 The reconstruction before and after the modification in the dataset

- The inital dataset specifications were:
  100000 Images
  2 Shapes in each image (total of 6 shapes)
  3 Operations in each image
- The dataset had a large proportion of dark images. This resulted in the loss of information on color in the latent space of FactorVAE*Kim and Mnih [2018]*. From our findings, we restricted the hue of the shapes that were being generated. To fix this issue we constrained the camera and directional light to face the shape's combined forward transform.

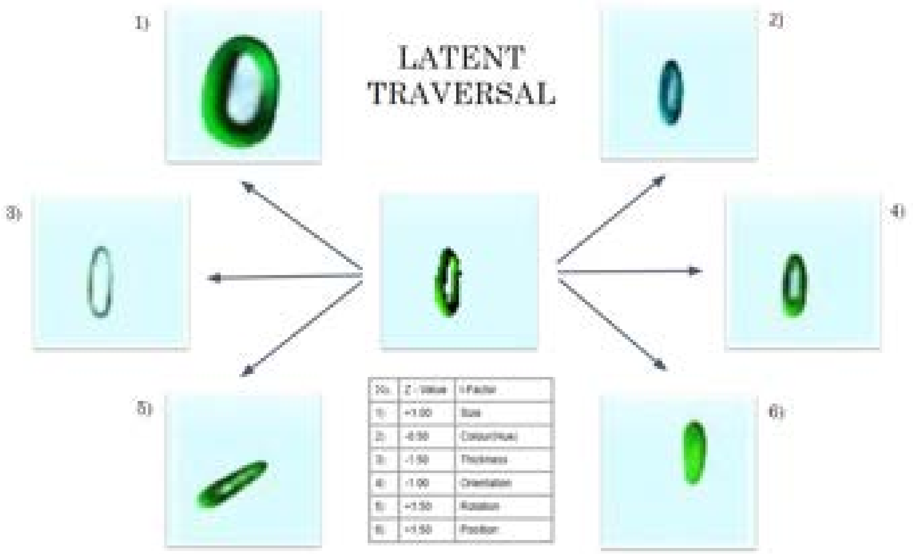
```
        Camera.main.transform.LookAt(this.transform);
        directional_light.transform.LookAt(this.transform);
```

- The initial version of the dataset covered 6 solid shapes. We incorporated more shapes such as the torus, that prevented the model from converging and essentially generalizing to a solid blob for every image in the latent space. The reconstruction before and after the modifications can be seen in Figure**??**.
- The final dataset specifications are:
  100000 Images
  4 shapes in each Image(17 shapes)
  3 possible operations

4.3 Representation Learning and 3D feature extraction

Modified Implementation of FactorVAE During the training of Factor-VAE, the permutation-invariant loss function is used to measure the independence between the factors. The approach enforces the independence between the factors by adding a Total Correlation (TC) penalty to the loss function. The TC penalty encourages the latent factors to be independent, making the optimization problem more difficult. For calculating the TC penalty. Factor-VAE randomly permutes the latent variables in each batch to give us Z' from the latent variable Z. A discriminator is then trained to distinguish z from Z' and contribute to the TC loss. TC loss component is calculated by computing the KL divergence between the aggregated posterior distribution over the latent (Z') and the product of the individual posterior distributions(Z). What we have modified to how we calculate the Z' vector. Instead of permuting along all the dimensions of z, even if we permute over just one dimension, we can allow for the reconstruction of Z'. We hypothesize that this allows the Discriminator to more accurately distinguishes Z from Z' while training.

Latent Traversal and Extraction of 3D Information



No.	Z - Value	i-Factor
1)	+1.00	Size
2)	-0.50	Colour(Hue)
3)	-1.50	Thickness
4)	-1.00	Orientation
5)	+1.50	Rotation
6)	+1.50	Position

Figure 5 Variation in the independent factors through the latent space

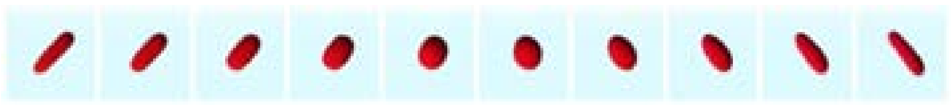


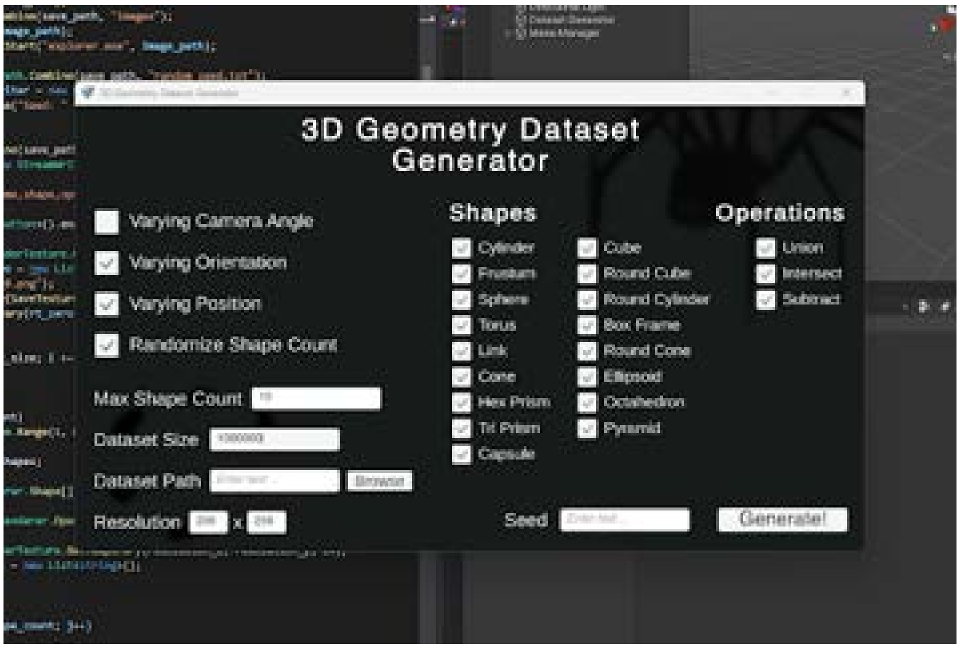
Figure 6 Gradual traversal across the latent space, z varies from -2.0 to +2.0

Once trained, we can traverse the latent space of FactorVAE to get variations across various factors of the shapes. This allows us to generate new shapes with variations in their features such as orientation, rotation, and displacement. Moreover, the disentangled representations learned by FactorVAE allow us to extract 3D information from the 2D image that was provided in the input. For instance, we show that by manipulating the latent factors, we are able to extract 3D features of the given shape from 2D images. This was possible because of the complexity of the dataset, comprising of various and varied shapes along with taking into account the constructive operations, intersection, union, and subtraction involved while preparing it.

The Figure5 shows the latent traversal through the latent space of the modified Factor VAE. The size of the latent space is (10*1), of which each entry corresponsd to the sampled input from the distribution presented by the encoder after refactoring. The figures show the generated image from the decoder for the given sample with the specified value for a particular factor. It can be observed that on explicitly varing the z value for a particular factor, lets say, orientation we can observe gradual changes in the shape's orientation. The fact that z is being sampled from a continuous distribution ensures that the changes are smoothened out, i.e gradual. This can be seen for a cylinder from Figure6.

This has applications in various domains, including virtual reality, gaming, and robotics. This has the potential to revolutionize the field of computer vision and provide new insights into the nature of generative factors in images.

5 Tool Release and Distribution


3D Geometry Dataset Generator

- [ ] Varying Camera Angle
- [✓] Varying Orientation
- [✓] Varying Position
- [✓] Randomize Shape Count

Max Shape Count: 10
Dataset Size: 1000000
Dataset Path: Enter text...
Resolution: 256 x 256

Shapes
- [✓] Cylinder
- [✓] Frustum
- [✓] Sphere
- [✓] Torus
- [✓] Link
- [✓] Cone
- [✓] Hex Prism
- [✓] Tri Prism
- [✓] Capsule
- [✓] Cube
- [✓] Round Cube
- [✓] Round Cylinder
- [✓] Box Frame
- [✓] Round Cone
- [✓] Ellipsoid
- [✓] Octahedron
- [✓] Pyramid

Operations
- [✓] Union
- [✓] Intersect
- [✓] Subtract

Seed: Enter text...

Figure 7 The Windows build of our tool.

- We developed an SDF Renderer in the Unity Game Engine.
- The renderer supports over thirty primitives, three operations (Union, Intersection, and Subtraction), and RGB color values (along with shadows).
- The source code for the renderer can be found at `https://github.com/aniketrajnish/CS499-SDFNet/tree/main/Renderer`.
- The engine uses signed-distance functions (SDFs) to render individual shapes using the shape index, dimension data, and RGB values. Further, it knits them into the screen space using their relative position and dimensions. 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.
- 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:

```
    case RaymarchRenderer.Shape.Pyramid:
        dim = new vector12(PyramidDimensions.size,
                           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
        return dim;

    case RaymarchRenderer.Shape.Triangle:
        dim = new vector12(TriangleDimensions.sideA.x,
                           TriangleDimensions.sideA.y,
                           TriangleDimensions.sideA.z,
                           TriangleDimensions.sideB.x,
                           TriangleDimensions.sideB.y,
                           TriangleDimensions.sideB.z,
                           TriangleDimensions.sideC.x,
                           TriangleDimensions.sideC.y,
                           TriangleDimensions.sideC.z,
                           0, 0, 0);
        return dim;
```

- Computer buffers were also used to communicate other information like the number of shapes to be rendered and their properties.
- All the shapes are rendered on render texture in front of the camera, the dimensions of which are communicated to the shader.
- The tool is built on top of this Raymarching Engine and is deployed as an executable (.exe) file.
- This system is designed to leverage the processing power of the GPU through direct computation and is optimized to achieve a constant time complexity.
- To prevent performance degradation when generating large datasets, the system employs a technique known as batch processing, which involves generating images in small groups (or batches), clearing the GPU memory, and repeating the process until all images are generated. This approach ensures that the performance of the system remains consistent, even when processing large datasets, thereby enabling efficient and effective image generation at scale.

```
        for (int i = 0; i < dataset_size; i += batch_size)
        {
            ...
            for (int j = 0; j < shape_count; j++)
            {
                ...
                Destroy(go);
            }
            ...
            RenderTexture.ReleaseTemporary(rt);
            file_names.Clear();
        }
        csvWriter.Close();
        ...
    }
```

- The tool and its source code will be made publicly available on the popular code hosting platform Github following the completion of Pacific Graphics 2023, a leading conference in the field of computer graphics and visualization. Meanwhile, it can be accessed from the following link: `https://drive.google.com/file/d/1OkwSgqqx2isK-lQN5Vudq4vGYJCSJ-ND/view?usp=sharing`

6 Conclusion and acknowledgements

In conclusion, the work presented in this article has introduced a comprehensive dataset of 3D shapes generated using SDFNet, a proprietary rendering tool. This dataset was created by manipulating primitive shapes using Boolean operations such as union, intersection, and subtraction, and is intended to be used for Constructive Solid Geometry algorithms and Disentangled Representation Learning. The availability of such a dataset has the potential to disrupt the current leading VAE-based disentanglement frameworks and facilitate the development of models that can generate low-poly 3D shapes, benefiting applications such as video game development and architectural design.

The proposed dataset and tool have been demonstrated to be effective in training FactorVAE to learn disentangled representations of the shapes in the dataset. The results of this study show that the dataset can be used to extract 3D information from 2D images by traversing the latent space of FactorVAE. This is a significant advancement in the field of disentangled representation learning, where the complexity of generative factors can hinder the ability of VAEs to disentangle the learned latent factors. The availability of a large and diverse 3D shape dataset, such as the one presented in this article, can provide the necessary complexity for training VAEs to extract 3D information from 2D images.

The dataset presented in this work overcomes the limitations of the existing datasets, which do not cover multiple shapes per image and are limited to only a few primitive shapes. The proposed dataset includes various primitive shapes, their dimensions, orientations, and color information, making it useful for training VAEs to extract 3D information from 2D images. Furthermore, the tool for generating personalized datasets can be used to create datasets for specific applications, making it a valuable resource for researchers and practitioners.

The availability of a large-scale dataset of low-poly 3D shapes will not only benefit the field of disentangled representation learning but also facilitate the development of applications that rely on 3D modeling, such as video game development, architectural design, and virtual reality applications. The proposed dataset and tool represent a significant step forward in this direction, providing researchers and practitioners with the necessary resources to advance the state of the art in these fields.

References

Leonidas Guibas Angel X. Chang, Thomas Funkhouser. Shapenet: An information-rich 3d model repository. URL `https://arxiv.org/abs/1512.03012`.

Federica Bogo, Javier Romero, Matthew Loper, and Michael J. Black. FAUST: Dataset and evaluation for 3D mesh registration. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, Piscataway, NJ, USA, June 2014. IEEE.

Chris Burgess and Hyunjik Kim. 3d shapes dataset, 2018. URL `https://github.com/deepmind/3d-shapes/`.

Yong He, Kayvon Fatahalian, and Tim Foley. Slang: Language mechanisms for extensible real-time shading systems. *ACM Trans. Graph.*, 37(4), jul 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201380. URL `https://doi.org/10.1145/3197517.3201380`.

Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3d mesh renderer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

Hyunjik Kim and Andriy Mnih. Disentangling by factorising, 2018.

Xin Wang, Hong Chen, Siao Tang, Zihao Wu, and Wenwu Zhu. Disentangled representation learning. *ArXiv*, abs/2211.11695, 2022.

A. Khosla F. Yu L. Zhang X. Tang Z. Wu, S. Song and J. Xiao. 3d shapenets: A deep representation for volumetric shapes.