Approximating Signed Distance Field to a Mesh by Artificial Neural Networks

Zhe Zeng
Purdue University

APPROXIMATING SIGNED DISTANCE FIELD TO A MESH BY ARTIFICIAL NEURAL NETWORKS

by

Zhe Zeng

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

Department of Computer Graphics Technology

West Lafayette, Indiana

May 2018

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Tim McGraw, Chair

Department of Computer Graphics Technology

Dr. David Whittinghill

Department of Computer Graphics Technology

Dr. Xavier Tricoche

Department of Computer Science

Approved by:

Dr. Bedrich Benes

Graduate Program Co-Chair

Dr. Colin M. Gray

Graduate Program Co-Chair

I would like to dedicate this thesis to my grandmother

ACKNOWLEDGMENTS

I wish to gratefully acknowledge my thesis committee members - Dr. Tim McGraw, Dr. David Whittinghill, and Dr. Xavier Tricoche - for providing valuable feedback and suggestions for this project. Special thanks to my advisor Dr. McGraw for insightful guidance and comments on the design of experiments and thesis writing. I also wish to thank my friends for encouragement throughout this research project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
D	Discriminative model
FCNN	Fully-connected Neural Network
G	Generative model
GAN	Generative Adversarial Network
RT	Ray Tracing
SDF	Signed Distance Field (Function)

ABSTRACT

Author: Zeng, Zhe. M.S.
Institution: Purdue University
Degree Received: May 2018
Title: Approximating Signed Distance Field to a Mesh by Artificial Neural Networks
Major Professor: Tim McGraw

Previous research has resulted in many representations of surfaces for rendering. However, for some approaches, an accurate representation comes at the expense of large data storage. Considering that Artificial Neural Networks (ANNs) have been shown to achieve good performance in approximating non-linear functions in recent years, the potential to apply them to the problem of surface representation needs to be investigated.

The goal in this research is to exploring how ANNs can efficiently learn the Signed Distance Field (SDF) representation of shapes. Specifically, we investigate how well different architectures of ANNs can learn 2D SDFs, 3D SDFs, and SDFs approximating a complex triangle mesh.

In this research, we performed three main experiments to determine which ANN architectures and configurations are suitable for learning SDFs by analyzing the errors in training and testing as well as rendering results. Also, three different pipelines for rendering general SDFs, grid-based SDFs, and ANN based SDFs were implemented to show the resulting images on screen.

The following data are measured in this research project: the errors in training different architectures of ANNs; the errors in rendering SDFs; comparison between grid-based SDFs and ANN based SDFs. This work demonstrates the use of using ANNs to approximate the SDF to a mesh by learning the dataset through training data sampled near the mesh surface, which could be a useful technique in 3D reconstruction and rendering. We have found that the size of trained neural network is also much smaller than either the triangle mesh or grid-based SDFs, which could be useful for compression applications, and in software or hardware that has a strict requirement of memory size.

CHAPTER 1. INTRODUCTION

1.1 Statement of Problem

Ray tracing is a computer graphics technique for generating images by simulating ray propagation in a 3D scene. It is widely used for rendering photo-realistic images by renderers like Mental Ray, for example, in applications such as indoor scenes, industrial design, and animation. However it is more expensive to render a complex mesh in ray tracing compared to other rendering technique like rasterization. Consider the problem of rendering a triangle mesh that has n triangles by a camera that shoots m rays as an example. In a trivial ray tracing algorithm, it must solve the intersections between m rays and n triangles, which makes the time complexity of the algorithm $\mathbf{O}(m \times n)$. Many methods have been presented to optimize it, for instance, querying the triangles using a designed binary tree by direction of the ray can certainly reduce the complexity to $\mathbf{O}(m \times \log n)$, but the coefficient could still be too large to render meshes with more triangles. Another example is rendering a surface by ray marching with a precomputed grid-based SDF. Unlike ray tracing a triangle mesh, the algorithm moves along the ray iteratively by the shortest distance to the surface. For any closed triangle mesh, the shorted distances can be represented as SDF using methods like level-sets which are precomputed before rendering. Since the time complexity for querying the precomputed data is $O(1)$ and the number of iterations is also a constant for a given bounded space, the complexity of this method is $\mathbf{O}(n)$, which is a good improvement compared to other methods described above. However, to make the rendering accurate, many samples of the shortest distance must be precomputed and stored, which makes the storage complexity $\mathbf{O}(x \times y \times z)$, where x, y, and z are the dimensions of the voxel grid.

Recently, researchers have been using ANNs to approximate non-linear relationships by learning from samples. For example, Ren et al. (2013) used a Fully Connected Neural Network (FCNN) to learn the Bidirectional Reflectance Distribution

Function (BRDF) and got a good results on reconstructing indirect illumination. So the possibility of learning the SDF to a mesh using ANNs is worthwhile to be explored.

1.2 Scope

The main focus of this research is to investigate if ANNs can be applied to the problem of approximating the SDF to a mesh. In this research, different architectures of ANNs were trained to learn different types of SDFs. Errors in training as well as rendering will be analyzed and discussed in the following chapters.

Topics covered in this research are ray tracing technique, ray marching technique with SDFs or grid-based SDFs, rendering triangle meshes, Generative Adversarial Networks (GAN) training and Convolutional Neural Netowrks (CNNs) training. Other surface representations, rendering techniques, methods of generating the SDF, and other ANN architectures are outside the scope of this work.

1.3 Significance

Ray tracing is a computationally-demanding technique for rendering photo-realistic images in 3D computer graphics. One approach to optimizing it is to use approximations rather than exact computations. ANNs are known to be capable of approximating a continuous function with a single hidden layer (Csáji, 2001). Although many methods like grid-based SDF have been used to do approximation in the past few years (Bærentzen & Aanæs, 2002), the possibilities of applying ANNs to the problem still need to be explored. Moreover, many previous researches in the computer graphics area have already used ANNs to, for example, do relighting of an image in screen-space (Ren et al., 2015). So it is useful to explore how ANNs can help solve problems in 3D space. Specifically, in this research, the ANNs can learn the SDF to a triangle mesh only through some samples near its surface, the dataset of which is much easier to prepare than other approaches that will be presented later. Also, the ANN based SDF can still be used to as a general SDF, consequently, it does not have problem in rendering transparent surface which is difficult for the methods from some previous works.

1.4 Research Question

The questions of research in this study is:

- Can ANNs approximate 2D signed distance functions?
- Can ANNs approximate 3D signed distance functions?
- Can ANNs approximate the signed distance field to a mesh?

1.5 Assumptions

This research is done using the following assumptions:

- The ray marching approach with signed distance functions can create a plausible rendering of a surface
- Triangle meshes used in the research are closed
- Level-set method can generate a correct SDF to a closed triangle mesh

1.6 Limitations

The following aspects limit this research:

- Large scene will not be tested in this research
- Complex shading, lighting, and unrelated optimization will not be considered
- SDF applies only to closed surfaces
- ANNs must be retrained if the SDF changes

1.7 Delimitations

The delimitations for this study include:

- This research will not investigate all types of distance functions
- Training and rendering data will be collected under specified environment
- This research will only test with specified types of triangle meshes and the distance functions
- Since this is a modeling approach, rendering time is not measured

1.8 Definitions

We define the following terms which appear throughout this thesis:

Artificial Neural Network: Artificial Neural Networks (ANNs) are a form of connectionism that can learn to do tasks by considering examples without any human programming (Medler, 1998).

Generative Adversarial Network: Generative Adversarial Networks (GANs) are a form of ANN introduced by Goodfellow et al. (2014), which are implemented by a system of two ANNs under a contesting zero-sum game condition.

Ray Marching: Ray marching is an algorithm to approximate the intersection between a viewing ray and surface. In ray marching, the ray is cast from the eye then marches step by step. When the ray is within an acceptable distance threshold from the surface, the algorithm considers it as an intersection.

Signed Distance Function: Signed distance function takes coordinates of a point as input and returns the shortest distance between the point and a surface. The sign of SDF indicates whether the point is inside or outside the surface. Signed distance functions can be efficiently rendered using ray marching algorithm.

Grid-based Signed Distance Field: Grid-based Signed Distance Field is a discretized SDF stored on a voxel grid. For each voxel in the grid, the shortest distance between the voxel and the surface is computed and stored. Grid-based SDF is generated using a level-set method.

Triangle Mesh: Triangle mesh is a commonly-used data structure in 3D computer graphics to represent surfaces. In a triangle mesh, surfaces are connected by triangles with shared vertices and edges that can represent shape a wide range of models, for example, animals, plants, terrains.

1.9 Summary

This chapter introduced the research by outlining the statement of the problem, research questions, significance of this work, definitions of key terms, assumptions, limitations, and delimitations. In the next chapter, a review of relevant literatures will be provided.

CHAPTER 2. REVIEW OF RELEVANT LITERATURE

This chapter provides a review of existing studies that are relevant to this research.

2.1 Artificial Neural Networks (ANNs)

An ANN is a form of connectionism that can learn to do tasks by considering examples without any human programming (Medler, 1998). ANNs recently become a very active area of research, though the history of it can be traced back before the invention of computers. In 1943, McCulloch and Pitts (1943) developed the first model of ANN and tested logic functions like a or b on it, which is still a popular test case. In 1969, Minsky, Papert, and Bottou (2017) mathematically demonstrated that an ANN with only an input layer and output layer can learn limited things. Although some major innovations like multi-layered NNs, which are called deep today, made it possible to solve complex problems, there was limited knowledge about how to train it at that time (Rumelhart & McClelland, 1988).

Later, a method called 'back-propagation' for training deep ANNs was proposed by Rumelhart, Hinton, and Williams (1986), and it has become a very popular training method. In 2012 a project based on it beat the most advanced Image Recognition (IR) system at that time Krizhevsky, Sutskever, and Hinton (2012). Moreover, the revolution of computing ability in both software and hardware has made complex mathematical calculations much faster, which has led to benefits in training large ANNs.

A simple neuron, as shown in Figure 2.1, takes the input of a vector with *n* dimensions and forwards one output. The middle node, which is called neuron, has two types of operation: training and using. In training operation, the neuron is desired to learn a specific pattern in the input data. In using operation, the neuron will only forward the output while itself not being affected.

After years of research, there are many different ANN architecture - ways of arranging and connecting groups of neurons. Two typical types of architectures are listed below:

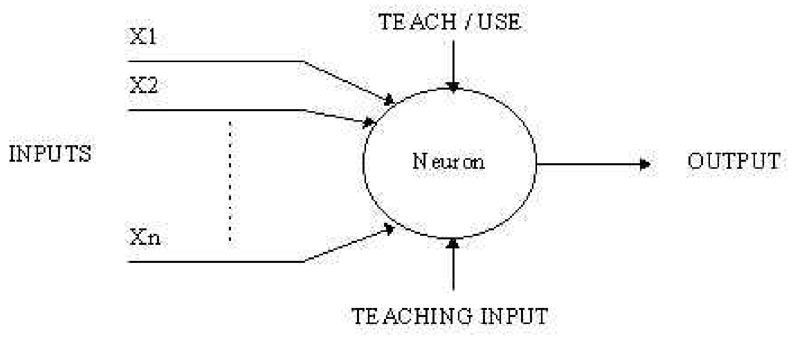


Figure 2.1. A Simple Neuron (Stergiou & Siganos, n.d.)

- Feed-forward Neural Networks (FFNNs). As the name indicates, such networks only have one direction, which means the input tensor is directly forwarded to output layer, and the result never affects the same layer. A typical architecture of FFNN is shown in Figure 2.2, which happens to be a fully-connected Neural Network (FCNN).
- Feedback Neural Networks (FNNs). Compared to FFNNs, nodes in feedback neural networks may have feedback loops from one layer to a previous layer.

The number of layers is another important characteristic of ANNs. A deep neural networks, which has multiple hidden layers, is better at prediction than ANN with single input and output layers (Svozil et al., 1997). A basic ANN usually has an input layer, an output layer, and multiple layers between them, which are called ’hidden layers’. In an ANN, each layer consists of multiple nodes with a binding weight. A network is formed when nodes in one layer are connected to the nodes in a neighboring layer.

Recently, ANNs have been widely applied to problems in computer vision, artificial intelligence, mathematics, statistics as well as computer graphics. In 3D computer graphics, ANNs have been used to do global illumination in real-time while achieving the same quality of rendering as offline rendering. For example, Ren et al. (2013) proposed a method of using an FCNN to approximate radiance regression functions of global illumination. This research trained a FCNN to learn the nonlinear coherence in the indirect illumination data while keeping the FCNN both compact and fast

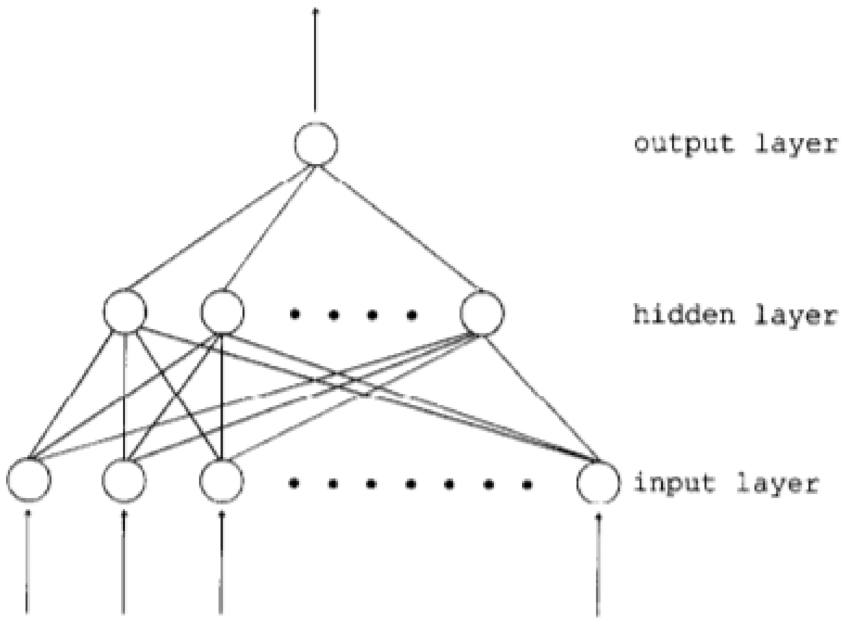


Figure 2.2. Feed-forward Neural Network (Svozil et al., 1997)

to use. Researchers fed the network data that were precomputed offline in ray tracing, which helped the FCNN learn to do similar illumination in real-time for the same scene. Their research finally achieved similar appearance to offline rendering, and the time performance was acceptable for being used in real-time rendering (Ren et al., 2013).

The rendering pipeline of their work is based on a multi-pass deferred shading method. In the first pass, the scene is rendered using a general rasterization algorithm. Then in the second pass, the indirect illumination can be predicted per pixel by FCNN with the existing G-buffers from the first pass including normals, texture colors, light directions, and view directions. The final image is generated by combining the two passes together (Ren et al., 2013).

The equation learned by FCNN is derived from reflectance equation (Cohen & Wallace, 2012):

$$s^t = s_a^t(x_p, v, l, n(x_p), a(x_p)) \tag{2.1}$$

Where s^t is the color channel of the indirect illumination, x_p is the surface point, v is the view direction, l is the light position, $n(x_p)$ is the normal at x_p, and $a(x_p$ is the texture mapped color at x_p.

Furthermore, based on this work, Ren et al. (2015) presented another method to do relighting on real photos. Similar to their first work, the idea behind it is to let FCNN to approximate light transport matrix **M** with the input of incident radiance **L** and outgoing radiance **I**. The equation can be presented as Ng, Ramamoorthi, and Hanrahan (2003):

$$\mathbf{I} = \mathbf{ML} \tag{2.2}$$

To obtain more coherent relationships, the input 4D vector is augmented with the average color of a pixel over all the captured images. The difference from their previous work is they trained multiple FCNNs for different subsets of the input images in an Adaptive Fuzzy Clustering (AFC) method. As shown in Figure 2.3, the AFC consists of three clusters, which stands for three base ANNs, and all pixels in the image are distributed to one or more FCNNs with the nearest distance. For testing, if a pixel drops to several FCNNs, which the researchers named 'NN ensemble', the result is the average of the ensemble. The datasets used in this research were generated by capturing 200 photos of the Toolset scene, 300 photos of the Horse scene, and 400 photos of the Indoor scene with a real camera and real light sources. Then the researchers train each FCNN in the similar manner to their previous work (Ren et al., 2015).

For another example, Tompson, Schlachter, Sprechmann, and Perlin (2016) presented a method using CNNs to accelerate Eulerian fluid simulation. The idea behind it is to train the ANN to learn Poissons equation, and apply conjugate gradient method to solve for pressure projection.

Even before ANNs were applied in computer graphics, a similar early theory called data-driven rendering (DR) approaches were used. For example, Blanz and Vetter (1999) presented a method to reconstruct 3D face model from a simple photo of face based on a database of 3D faces. For another example, Sato, Morita, Dobashi, and

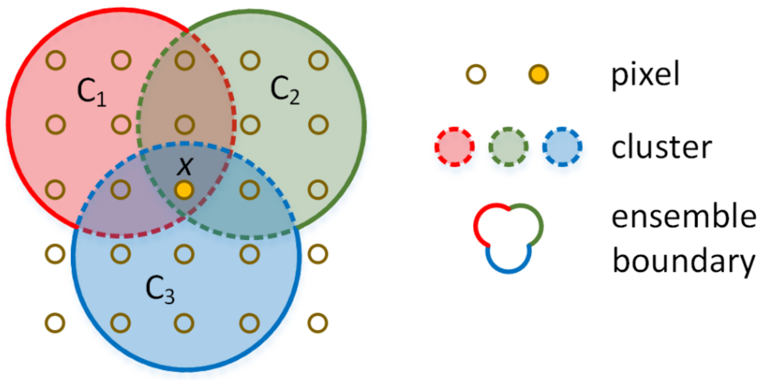


Figure 2.3. This AFC consists of three nasic NNs, each pixel in the image is assigned to the NN with nearest distance (Ren et al., 2015)

Yamamoto (2012) presented a model that can add details to low resolution fire with the data of high resolution fire data.

Compared to approaches listed above, this research focuses more on exploring hidden relationships in 3D space by letting FCNNs, GANs and CNNs earn general SDFs and SDFs to a complex mesh.

2.2 Convolutional Neural Networks (CNNs)

Like most other ANNs, CNNs are a class of neural network that are trained with back-propagation algorithm. The difference is CNNs use a variation of multilayer perceptrons inspired by biological processes to recognize the pattern from pixel image with minimal preprocessing (LeCun et al., 2015; Matsugu, Mori, Mitari, & Kaneda, 2003). Similar to a FCNN, a typical CNN consists of an input layer, multiple hidden layers, and an output layer, but the hidden layers consist of convolutional layers rather than fully connected layers (LeCun, Bengio, et al., 1995). Figure 2.4 shows a schematic of a very deep CNN named VGG-16, which is known for achieving top 5 test accuracy in ImageNet (Simonyan & Zisserman, 2014).

In recent years, CNNs have been used in object detection, natural language processing, and games (Maturana & Scherer, 2015; Shen, He, Gao, Deng, & Mesnil,

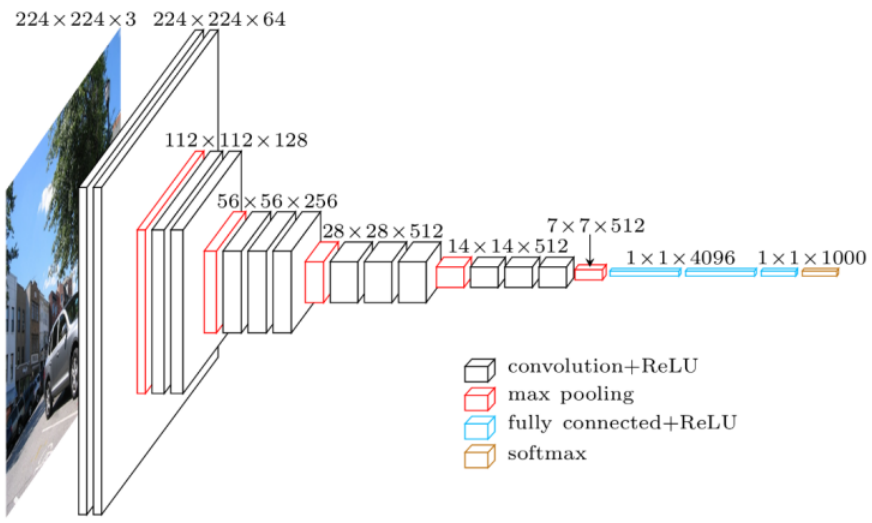


Figure 2.4. Architecture of VGG-16 (Blier, 2016)

2014; Silver et al., 2016). In computer graphics, Nalbach et al. (2017) proposed a method of doing GI with a U-shaped network architecture, also known as U-net, CNN. The idea behind their work is to let the CNN learn how to synthesize images of required GI effects from data in deferred shading buffers. Different from what Ren et al. (2013) did, they trained the ANN to learn a target equation. The researchers focused on 2D screen space directly by taking advantage of the CNN. As mentioned in their paper, approximating the appearance from attributes of the image is actually a reverse process of semantic segmentation, which was demonstrated in by previous works in computer vision area (Long, Shelhamer, & Darrell, 2015; Nalbach et al., 2017).

Nalbach et al. (2017) implemented U-Net fully convolutional networks (FCNs) for their experiments, which are widely used in biomedical image segmentation area (Long et al., 2015). FCN is a type of CNN that replaces fully connected layers at the end of the architecture by convolution layers, which is better at doing dense predictions for per-pixel tasks. In their case, as shown in Figure 2.5, the architecture of an FCN consists of an input layer, hidden layers, and an output layer. The input takes inputs from deferred shading buffers and the output layer generates the synthesized image. The hidden layers consist of two parts, in the left branch the spatial resolution of inputs are reduced level by level with a 2×2 mean pooling, in the right branch the resolution is recovered by a bilinear up-sampling. In each level of the branch, an activation function of leaky ReLU are used. Screen space effects including ambient occlusion (AO), depth-of-dield (DoF), and anti-aliasing were examined in their experiments. $61,000 pairs$ of deferred shading buffers and label images were used for training, validation, and testing. Different resolution of images were applied for different experiments, for example, images with $512 \times 512 pixels$ were used for AO, and $256 \times 256 pixel$ for all other effects. In total, $54,000$ images were used to train, $6,000$ for validation, and $1,000$ for testing. The images for training and for validation shared the same set from 10 scenes, while images for testing came from 4 unseen scenes. These images were sampled randomly from the scenes captured by a camera with a fixed field-of-view (FoV). The deferred shading buffers were rendering using rasterization, the label image were rendered using ray tracing. A special loss function based on structural similarity (SSIM) index Zhao, Gallo, Frosio, and Kautz (2015) was used for training, which compared corresponding $8 \times 8 px$ patches from the output to the ground truth. The structural dissimilarity (DDSSIM) was used as final loss, which was $(1 - SSIM)/2.0$. The weights were updated by a general stochastic gradient descent (SGD) algorithm. The results of their research accurately reproduced most screen-space effects and have acceptable real-time rendering speed performance. One drawback in the work is the training time is very long even with very low resolution. Another problem is that the training datasets are hard to prepare. Additionally, no screen space effects depending on transparency, like refraction or fresnel reflection were provided Nalbach et al. (2017).

1D CNNs are also explored as one of the main approaches in this research to approximating 2D SDFs. A comparison between the performance of different depths of CNNs as well as the performance between CNNs and other frameworks are explored in this thesis.

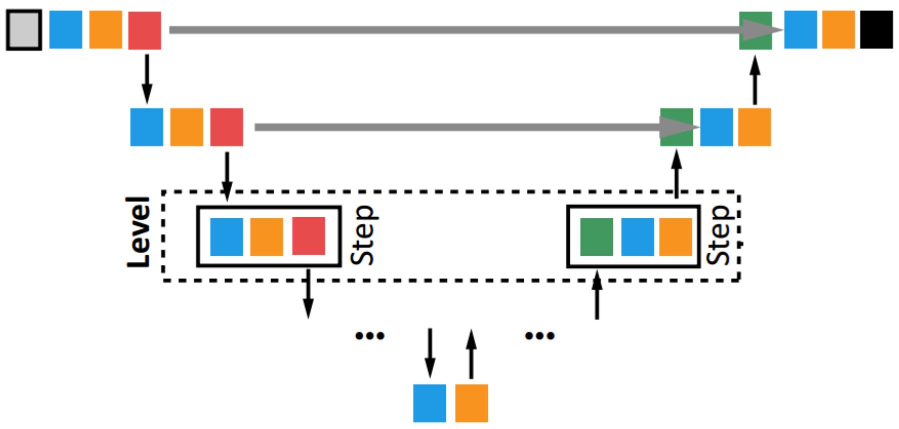


Figure 2.5. U-Net FCN for GI. Input block is in grey, output block is in black. Resolution of imaged are reduced in the left and recovered in the right. (Nalbach et al., 2017)

2.3 Generative Adversarial Networks (GANs)

The framework of GANs was proposed by Goodfellow et al. (2014), which introduced a new method to estimate generative model through an adversarial process. The framework is a system of two ANN models: a generative model *G* that captures the data distribution and a discriminative model *D* that is used to judge the training procedure for *G*. Both *G* and *D* are trained simultaneously corresponding to a zero-sum game where *G* is trained with the response from *D* and *D* is trained to learn how close the output from *G* is to ground truth. The previous frameworks of ANNs achieved striking success in mapping high-dimensional and rich sensory inputs to a class label (Hinton et al., 2012; Krizhevsky et al., 2012) while having difficulty in approximating many intractable computations like maximum likelihood estimation by taking advantage of the generative context (Goodfellow et al., 2014).

GANs have been used to generate natural images, super resolution, and to synthesize complex textures (Denton, Chintala, Fergus, et al., 2015; Ledig et al., 2016; Ulyanov, Lebedev, Vedaldi, & Lempitsky, 2016). In 3D computer graphics, Thomas and Forbes (2017) used a GAN to synthesis globally-illuminated images based a directional lighting buffer and G-buffers including the depth map, the color map, and the normal map.

In their research, system of the GAN consists of a U-net CNN as the *G* and a CNN as the *D*. Figure 2.6 shows the schematic of their *G* model (top) and *D* model (bottom). The *G* consists of an input layer that is fed by G-buffers and directional lighting buffer, eight convolutional layers with leaky ReLu as activation function, eight deconvolutional layers with ReLu as its activation function, and an output layer that generates the result. The *D* network consists of an input layer which takes G-buffers, directional lighting buffer, and ground truth or generated images from *G* Model as inputs, five convolutional layers with leaky ReLu as the activation function, and the output layer followed by the sigmoid function. From the figure, some details of indirect illumination as well as soft shadows can be clearly observed from their synthesized image (Thomas & Forbes, 2017).

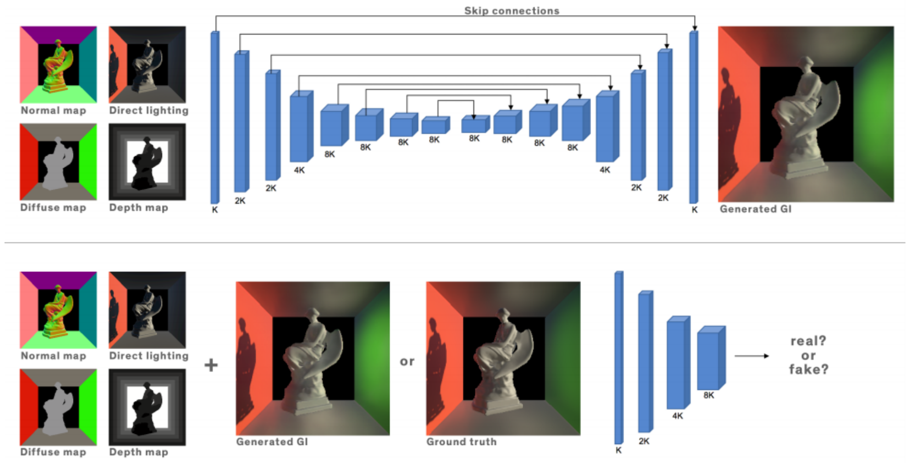


Figure 2.6. Generative Adversarial Networks Used to Approximate Global Illumination (Thomas & Forbes, 2017)

GANs are also applied as one of the main approaches in this research to approximating SDFs due to their good ability to learn data distribution from generated results, which could be helpful because the inputs of SDFs are low-dimensional with less explicit information.

2.4 Ray Tracing

Ray tracing (RT) is a rendering technique which traces the propagation of light rays in a scene. RT is widely used in many areas like computer vision, computer graphics, and telecommunication. The first ray tracing algorithm in computer graphics was presented by Appel (1968). Compared to ray launching algorithm, RT only needs to track the rays that hit the image plane of a virtual camera, which reduces a large amount of the computational effort.

Figure 2.7 shows how RT works. Typically, multiple rays are shot from the camera to each pixel on the image plane, and then into the scene. For each ray, the algorithm will compute the intersection with the surface. If the ray does not hit anything, a default color will be set for the pixel. If it hits a surface, the ray may bounce along another direction, which depends on the normal at the hit point and the maximum bounce count. Before bouncing, the color of the ray will be updated by the color of the hit point.

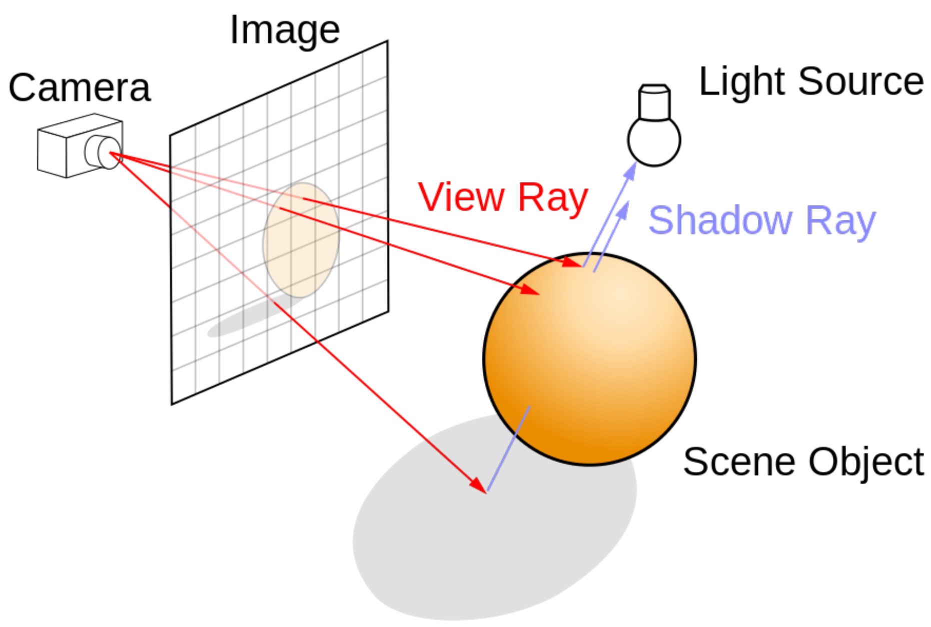


Figure 2.7. The ray tracing algorithm (Henrik, 2008)

Because it is an algorithm for simulating ray propagation, it can also be used to simulate light in the real world, which is one of the main reasons RT is often used in rendering realistic phenomenon over other rendering methods such as scan-line rasterization. Besides, it is simpler in RT to implement some complex effects than in rasterization. For example, in rasterization, shadows are often computed using shadow maps, which require the scene to be render from the light position to a depth map, also known as the shadow map. Then for each valid pixel of the view camera, the scene coordinates are transformed to light-space for depth testing. However, in ray tracing, it is very straightforward to do this by using a shadow ray. As shown in Figure 2.7, a ray is cast from the position of the valid pixel to the light source. If the ray hits any surface in the scene, the pixel is in shadow.

Regardless of these advantages, ray tracing algorithm has higher computational requirements than rasterization. Because in rasterization, many computations can be shared for coherent data, while ray tracing needs to do it from scratch for any new ray. These disadvantages make ray tracing more popular in industrial design, animation, and visual effects where images can be rendered slowly, as opposed to 3D games, and virtual reality (VR) which have strict real-time requirements.

In computer graphics, researchers have proposed many new techniques to optimize ray tracing. The first approach for real-time ray tracing was developed by Mike Muuss in 1986, which achieved an impressive performance at that time for multiple frames per second. Since then, many projects like OpenRT, NVIDIA's OptiX , and OpenRL were released as well as some special-purpose hardware like a ray processing unit for intensive operations developed by researchers at Saarland University.

A common approach in these projects is to optimize intersection computation between rays and a mesh. For instance, Carr, Hoberock, Crane, and Hart (2006) used a threaded bounding volume hierarchy built from a geometry image to quickly determine intersections. Another example is an algorithm presented by Möller and Trumbore (2005) to find ray-triangle intersections by using a special ray constructed from the original ray.

2.5 Triangle Mesh

This section is organized as follows: a brief review of the data structure of triangle meshes, and a review of a conventional algorithm of solving ray-mesh intersection.

The triangle mesh is a common type of mesh in 3D computer graphics. In triangle meshes, surfaces are connected by triangles with shared vertices and edges that can represent shapes for a wide range of models, for example, animals, plants, terrains, as shown in Figure 2.8. One advantage of such a data structure is that it can save space for storing shared vertices as well as save time for operating on them. It is useful when the mesh is very large with many faces and shared edges.

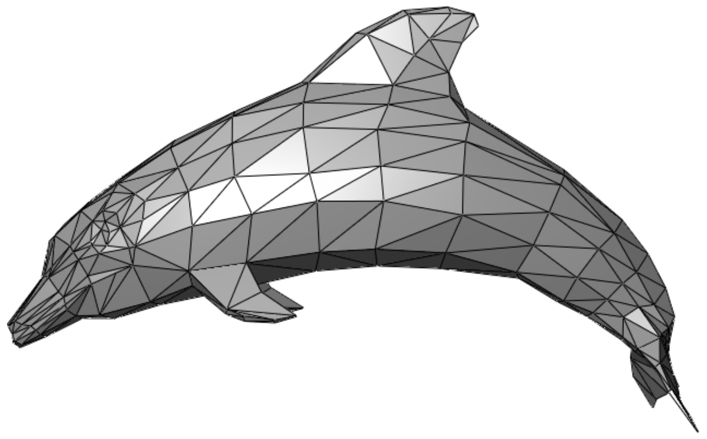


Figure 2.8. Triangle Mesh for Dolphin (Chrschn, 2007)

Many standards have been proposed by different organizations for storing triangle meshes, for example, Wavefront OBJ (.obj), Autodesk FBX Format (.fbx), and Polygon File Format (.ply). Most of them are encoded in binary or American Standard Code for Information Interchange (ASCII). A basic triangle mesh file consists of position coordinates, color value, texture coordinates, normal vector, number of vertices, number of triangles, and path of textures.

Triangle meshes are also popular in ray tracing, and many algorithms are designed to support them. For example, one indirect use of it is deriving SDF from the mesh, then using the SDF in rendering, which will be discussed in more detail later. The direct use of the mesh is computing the ray-mesh intersection. To find the intersection, a naive method could be calculating the intersection between the ray and all the triangles in that mesh (Appel, 1968). A better algorithm used for computing intersection between ray and triangle is the Mller-Trumbore (MT) algorithm Möller and Trumbore (2005). Although the MT algorithm was presented 20 years ago, it is still used as a benchmark for many optimization algorithms. In the MT algorithm, barycentric coordinates are used for points on the triangle, which satisfy the following equation, where P is the coordinate of the point on triangle, and A, B, C are the vertices of the triangle:

$$\mathbf{P} = u \times \mathbf{A} + v \times \mathbf{B} + w * \times \mathbf{C}. \tag{2.3}$$

If the hit point is inside of the triangle, then u, v, and w must be in the following interval:

$$0 \leq u, v, w \geq 1 \tag{2.4}$$

And satisfy the following equation:

$$u + v + w = 1 \tag{2.5}$$

Additionally, points on a ray satisfy the following equation, where O is rays origin, D is its normalized direction, and t is how far the ray is traversed:

$$\mathbf{P} = \mathbf{O} + t \times \mathbf{D}. \tag{2.6}$$

Consequently, the intersection between them can be defined as a system of linear equations, where t, u, and v are the solution of intersection:

$$\begin{bmatrix} -\mathbf{D} \\ \mathbf{B} - \mathbf{A} \\ \mathbf{C} - \mathbf{A} \end{bmatrix} + \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{B} - \mathbf{A} \tag{2.7}$$

In ray marching, intersection can be calculated indirectly by SDFs, which will be reviewed in the next section.

2.6 Ray Marching and Signed Distance Functions

This section reviews literature related to ray marching and signed distance functions.

2.6.1 Ray Marching

Ray marching is a type of rendering technique based on ray casting algorithm used to compute intersection between a ray and surface. Very similar to ray tracing, rays are sent from camera through the image plane into the scene. A main difference between them is how the intersection is found. Typically, for each path tracing, the ray tracing technique computes the intersection between the ray and the surface directly, while in ray marching, a point is moved along the ray direction step by step. At each step it is determined whether the point is on the surface based on the distance to the surface.

Ray marching has been applied to render photo-realistic images and screen-space effects for multi-pass rendering. For example, Zhou et al. (2008) presented an algorithm named compensated ray marching for rendering smoke in real-time.

2.6.2 Signed Distance Function

In ray marching, SDF is used to get the shortest distance from a voxel to the surface as well as determine if the voxiel is inside or outside the surface. Ray marching can render SDF to a surface very efficiently. Figure 2.9 shows how a point on the ray approaches the surface. Firstly, for P_0, the shortest distance is computed by the SDF, which is the distance between P_0 and the top vertex of the triangle at the bottom, so the point will be moved to the position of P_1, and because P_1 does not hit the surface, similar moves and checks are repeated from P_1 through P_4. Finally at P_4, the marching could stop as it hits the surface. Surface intersection is assumed to happen when the distance to the surface is less than a small positive threshold to make the algorithm run efficiently, .

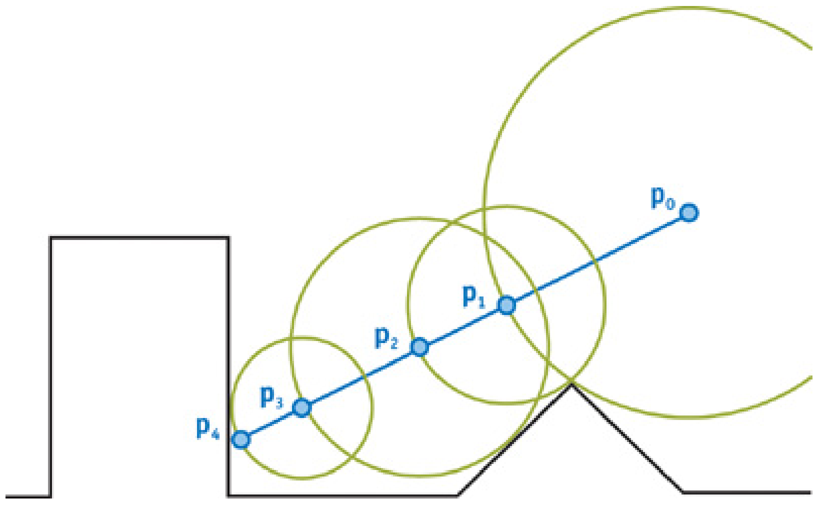


Figure 2.9. SDF in Ray Marching (Donnelly, 2005)

Similar to ray tracing, explicit formulas can also be used as SDF for modeling either 2D or 3D shapes. For example the shortest distance from *p* to a sphere is the distance between *p* and the sphere's origin minus the radius. And for a union of two shapes, the shortest distance the shorter distance of each shape. As shown in Figure 2.10, a scene consisting of multiple shapes can be rendered by a map of similar formulas.

In this research we intend to train ANNs to learn both 2D SDFs and 3D SDFs. For all 2D SDFs and part of 3D SDFs, the datasets are generated using explicit formulas. For learning 3D SDFs to a triangle mesh, the dataset is generated from a grid-based SDF created using level-set methods, which is dicussed in the next section.

2.7 Level-set Method and Grid-based Signed Distance Field

The level-set method is a technique for using implicit functions to do numerical analysis of surfaces and shapes, which was proposed by Osher and Sethian (1988). The technique can easily track the shapes by using a fixed Cartesian grid without parameterizing them. In the grid, the closed curve Γ is represented by a level-set function

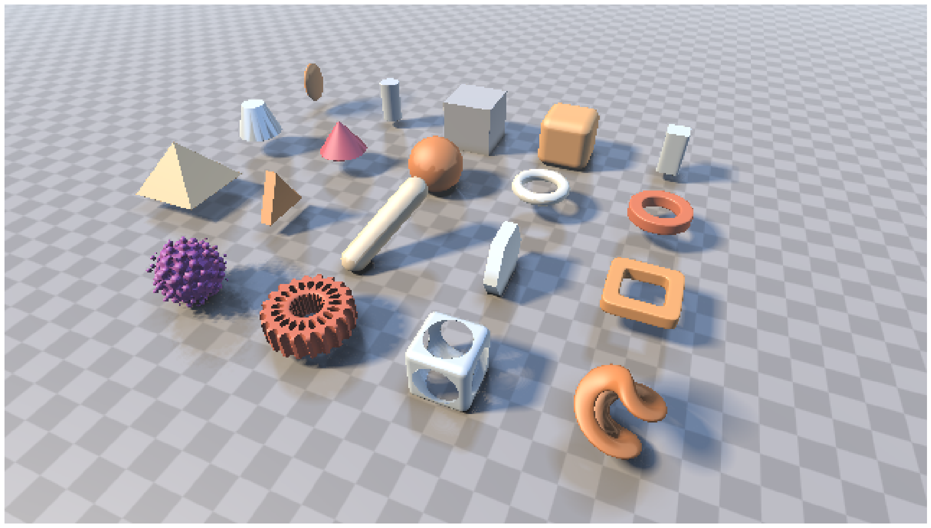


Figure 2.10. Primitives in Ray Marching by Basic SDFs (Quilez, 2013)

φ when $\varphi = 0$. Moreover, the value of φ for points outside of Γ is negative, for points inside of Γ is positive Osher and Sethian (1988).

The level-set method has been used in many computer vision and computer graphics applications, including image segmentation, simulation of complex water surface, volume segmentation, and shape modeling (Bærentzen & Christensen, 2002; Enright, Marschner, & Fedkiw, 2002; Malladi, Sethian, & Vemuri, 1995; Whitaker, Breen, Museth, & Soni, 2001). The technique is not varied too much when applied to the problem of generating the SDF to a shape. Such methods often involve a scan conversion step for calculating the signed shortest distance to the surface for each voxels in the grid.

Bærentzen and Aanæs (2002) proposed a different level-set based method without scan conversion to generate SDF from a triangle mesh. The first step of their method is calculating the shortest distance for each voxel and recording information of the closet point. The shortest distance is computed by simply comparing the distances between the voxel to all triangles of the mesh. The sign is determined in the second step, if the stored closest feature is a vertex or an edge, a precomputed angle weighted normal (Séquin, 1987) will be used to determine whether a voxel is inside or outside. If it is a face, then the face normal will be used (Bærentzen & Aanæs, 2002).

In the following experiments approximating 3D SDF to a mesh, the grid-based SDF technique is used to compare between ANN approaches. The grid-based SDF is generated from a triangle mesh of bunny.

2.8 Summary

This chapter provided a review of the literature that is relevant to this research. Ray tracing is a rendering technique that casts a ray from the eye and computes the intersection with the scene. The corresponding pixel that the ray passed through is usually drawn with color of this ray. Many previous algorithms are presented to improve this process, including hardware acceleration and precomputation. This research is about training ANNs to learn SDFs for use in ray marching. The goal is to learn the relationship between points and nearest distance to a surface in bounded 3D space.

The next chapter provides detailed methodology used in this research.

CHAPTER 3. FRAMEWORK AND METHODOLOGY

This chapter introduces the framework of the application involved in this research, including the rendering pipeline and the experimental environment. Also, our hypotheses, procedures for the experiments, and description of the algorithms, variables, and measurements are included in this chapter.

3.1 Hypotheses

The hypotheses of this work are listed below:

- ANNs can approximate 2D signed distance functions
- ANNs can approximate 3D signed distance functions
- ANNs can approximate the signed distance field to a mesh

3.2 Procedures for the experiments

The experiments are designed as below:

1. Use multiple ANN architectures to learn 2D SDFs and compare the result to the ground truth
2. Apply the optimal configuration of ANN from previous experiment and assess the ability of the network to learn 3D SDFs and compare the result to the ground truth
3. Apply the optimal configuration from the previous experiment to the SDF to a triangle mesh and compare the result to the ground truth

3.3 Variables

This section introduces control variables, independent variables, and dependent variables used in the experiments.

3.3.1 Control Variables

Control variables used in the experiments are listed below:

- Configuration of the virtual camera used to display the surface
- The training data: SDFs for geometric surfaces and meshes
- Training epochs

3.3.2 Independent Variables

In these experiments, the independent variable is the input of the SDF, which is the coordinate of a pixel or voxel.

3.3.3 Dependent Variables

In these experiments, the dependent variable is the shortest distance between input coordinate and the surface.

3.4 Measurements

This section introduces how results in the experiments are measured.

3.4.1 Errors in training and testing

Training errors are collected during training and measured by the loss function of the ANN. Two types of testing errors are measured in this experiment. The first one is the error in all pixels on the image plane, the second is errors in rendering the surface, which means the background is exclusive. The errors are measured using mean squared errors (MSE) approach by calculating the distance between the ground truth and what is approximated by the ANN.

3.4.2 Storage cost

When the ANN can render SDF with an acceptable error, the storage cost for the ANN is compared to the storage cost of the discretized SDF grid generated by level-set algorithm. We also compare to the size of triangle mesh.

3.5 Dataset

For camera:

- 200×200 pixels
- Camera is positioned at $[0.0 \quad 0.0 \quad 0.0]$
- Camera has a view direction of $[0.0 \quad -1.0 \quad 0.0]$, and a horizontal field of view of 90°

For triangle mesh or SDFs:

- 2D SDFs consisted of geometric primitives like circles and rectangle, as well as more complex shapes formed by boolean operations like union and subtraction of 2D primitives.
- 3D SDFs consisted of geometric primitives like spheres and cubes, as well as more complex shapes formed by boolean operations like union and subtraction of 3D primitives.
- A triangle mesh of the Stanford bunny is used in the third experiment

3.6 Development Tools

The application is developed using the following tools:

- Python: Used to write the main application including the ray marching pipeline, grid-based SDF parser, and data visualization tools.

- PyTorch: Used to build the ANNs and manage the training and testing process.
- CUDA: Used to accelerate the training of the ANNs on the GPU.
- Python multiprocessing: Used to accelerate the testing part.

3.7 Rendering Pipeline

The application has different pipelines for different rendering situations:

- For rendering analytical SDF represented as a mathematical equation, the program first computes distance for each pixel in parallel, then outputs the rendering result
- For rendering grid-based SDF, the program firstly loads the SDF into memory using a custom parser. During rendering, the program looks up the data structure for the shortest distance, then outputs the rendering result.
- For rendering ANN-based SDF, the program first loads it using a built-in parser in PyTorch. During rendering, the program evaluates shortest distances using the ANN and outputs the result after computing all pixels.
- A bounding box is used to render the surface more efficiently. Rays which do not intersect the bounding box cannot intersect the surface, and are trivially assigned the background color.

3.8 Environment

The experiments are conducted under the following environment:

- Intel Core i7-3770 @ 3.40GHz
- 16 GB Memory
- Graphics NVIDIA GeForce 960

- Windows 10 Education 64-bit Operation System

3.9 Summary

This chapter provides an overview of the design of our experiments as well as the rendering pipeline of this work. Accuracy and storage are assessed in this experiment. The next chapter provides the results of these experiments.

CHAPTER 4. RESULTS

This chapter presents the results from the three experiments we conducted, including rendering results, errors in training, and errors in testing. In the 3D rendered results presented here, the final color is linearly shaded based on the depth of each visible pixel.

4.1 Training Process

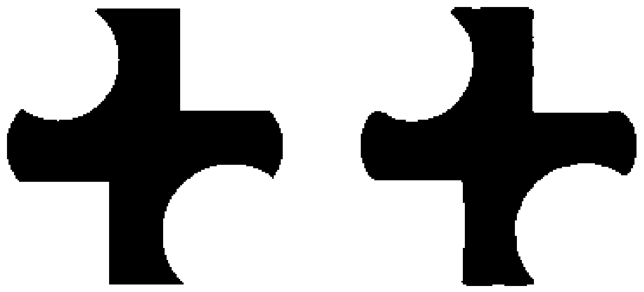
In Figure 4.1, the left side is the ground truth 2D SDF rendered by plotting negative SDF values in black (inside the boundary), and positive values are plotted as white pixels. The right side is rendered by GAN. Preliminary results on simple geometric shapes, such as circles and squares, revealed difficulty in representing regions with high curvature. This specific shape was created to test how well the ANN can reconstruct sharp corners.

Similarly, in Figure 4.2, the left side is the ground truth rendered with 3D SDF by ray marching while the right one uses a SDF represented by GAN. In Figure 4.3, the left side is the triangle mesh of a bunny (displayed using Paint 3D), the middle image is rendered with a grid-based SDF for the same mesh by ray marching, and the right side is rendered using the GAN SDF. High resolution rendering results are presented in the appendix.

Withing the three main experiments, many small experiments were conducted to determine the best architecture and configuration for ANNs to represent the SDF. In the following subsections, the most important findings are described.

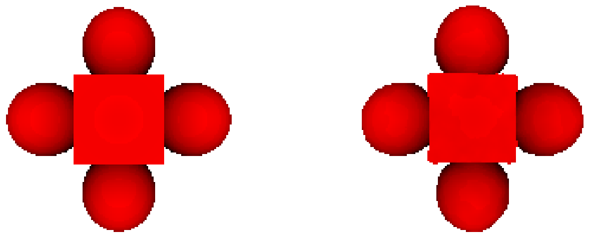
4.1.1 Frameworks of Models

Three different Frameworks of ANN models were assessed in these experiments: FCNN, CNN, and GAN. Since it is faster to train a network to learn a 2D SDF,



(a) Ground Truth (b) GAN

Figure 4.1. Results Rendered for 2D SDFs



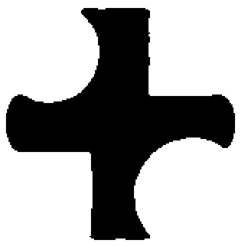
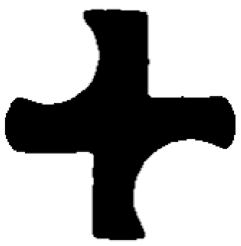
(a) Ground Truth (b) GAN

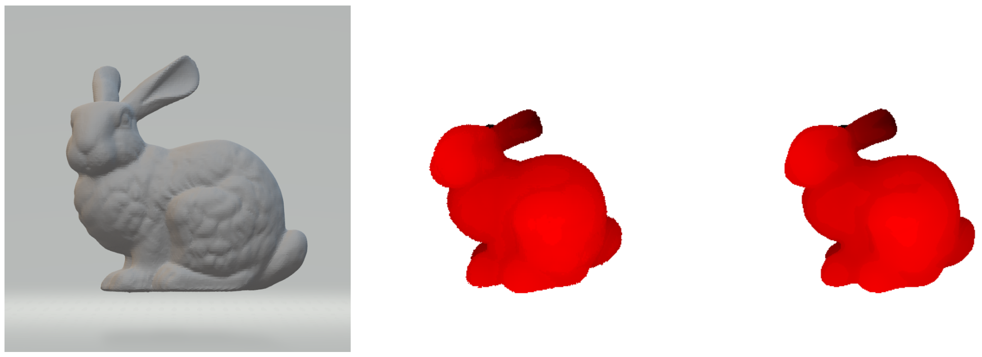
Figure 4.2. Results Rendered for 3D SDFs

preliminary experiments were conducted in 2D, and the most promising of those methods were then tested in 3D.

In the first step of the experiment on 2D SDFs approximation, FCNN was used to approximate a basic SDF to a circle, but the result was much worse than CNN. In the next step, CNN and GAN were used to approximate a more complex 2D shape. Although CNN performed well in this step, the GAN had smaller errors shown in the following Table 4.1. Consequently, only the GAN was used to approximate 3D SDFs. Additionally, for the parameters of network width and depth in the table, -1 denotes the number of nodes from last layer, which is determined by the convolutional layer and the max pooling layer.

Table 4.1. Rendering Errors of GAN with Uniform Sampling Dataset

#	Generative Model of GAN		CNN	
width and depth	linear layer	2×256	linear layer	2×256
	conv layer	1×16	conv layer	1×16
	conv layer	16×32	conv layer	16×32
	linear layer	-1×1	linear layer	-1×1
overall errors	3.05×10^{-5}		1.89×10^{-3}	
errors in pixels on the surface	1.88×10^{-5}		1.41×10^{-4}	
result				

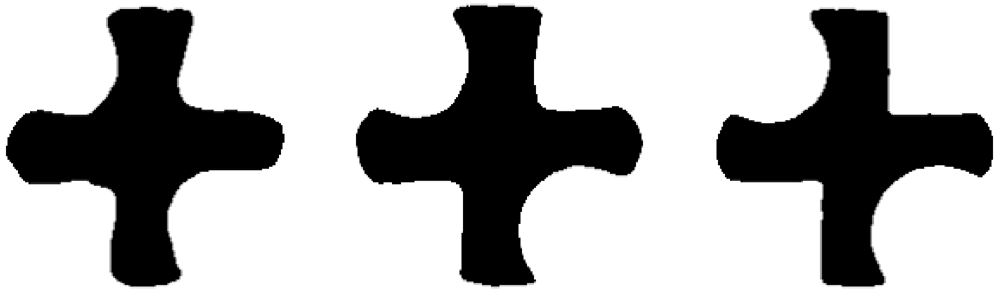


(a) Ground Truth (b) Grid-based SDF (c) GAN

Figure 4.3. Triangle mesh (a), discretized SDF computed from the mesh (b), and the surface reconstructed by GAN (c).

4.1.2 Width and Depth of the Model

Width, which is number of nodes in each hidden layer, and depth, which is the number of hidden layers, are also two important factors that have a major impact on the training results. Figure 4.4 shows how width of convolutional layer affects the rendering results. The output became significant when the width was increasedfrom 64 nodes to 256 nodes.



(a) 64 Nodes (b) 128 Nodes (c) 256 Nodes

Figure 4.4. Results for Complex 2D SDF

Figure 4.5 shows different rendering results for ANNs with different depths. As you can see, there is a significant improvement from ANN with 3 hidden layers to ANN with 4 hidden layers, but from 4 hidden layers to 5 hidden layers, the improvement is not very significant, but the curve of ANN with more hidden layers is more regressive.

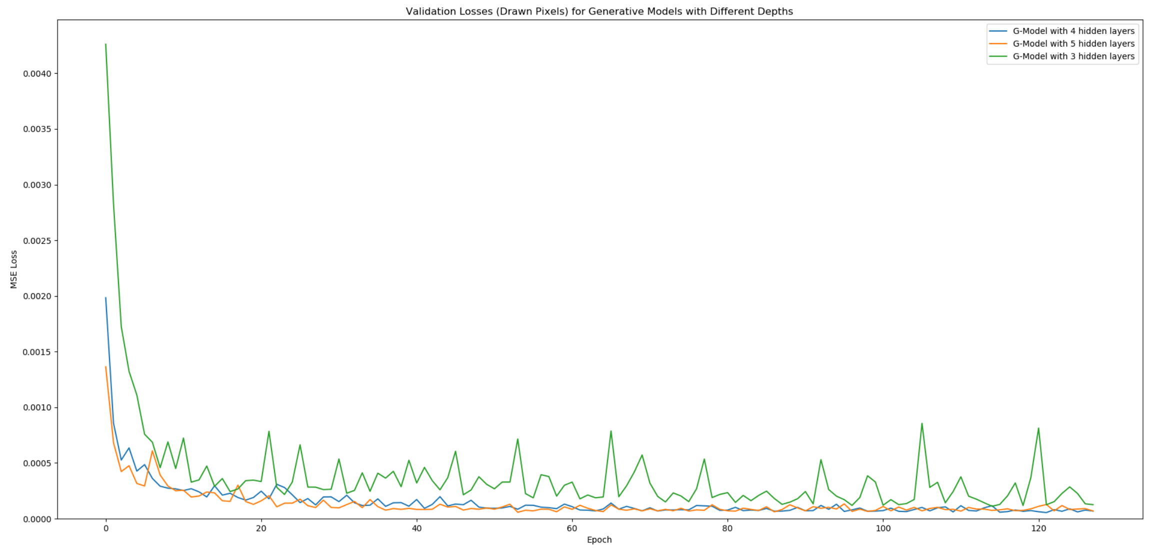


Figure 4.5. Errors in Validation for CNNs and GANs with Different Number of Hidden Layers

4.1.3 Dataset

The experiment initially used randomly generated coordinates ranging from -2.0 to 2.0 as inputs and the shortest distances calculated by SDF as labels to train the ANN, but we found out the result would be accurate only if these inputs are distributed uniformly. For example, in Figure 4.6(b), the bottom part of the circle was missed because the dataset only covers few inputs from that range. So one problem of this method is that if the randomly generated vectors fail to cover some geometric features, the object would be rendered incorrectly.

Consequently, we turned to another approach to do a uniform sampling for the dataset. Take sampling a 2D box as an example, as you can see in Figure 4.7, the bounding box of the 2D box is divided into 16×16 blocks along its horizontal axis and vertical axis, where the black bold line is its surface, and X represents the sampling points. For the uniform sampling approach, coordinates of all X are sampled, which can definitely avoid the problem mentioned in random sampling.

But one problem in uniform sampling is some details like sharp edges were not very accurate, although the mesh can be rendered well in general. So based on this observation, an improved approach that only samples the points near the surface, which is called narrow-band surface sampling, was applied. For example, in Figure 4.7, only points in red will be sampled. Far or near is determined by the absolute value of the signed distance to the surface. As a result, the rendering results by the ANNs trained with dataset using narrow-band surface sampling have better details than the other sampling approaches, as you can see in Figure 4.8.

However, in Table 4.2 for errors in approximating 2D SDFs, the fitting errors in ANNs trained with uniformly sampled datasets are lower than the one with narrow-band surface sampling. The main reason for this is the latter one is less accurate at predicting for the pixel or voxel far from the surface regardless of the better performance near the surface.

Table 4.2. Fitting Errors for ANNs with Dataset by Different Sampling Methods

#	uniform sampling	narrow-band surface sampling
overall errors	5.44×10^{-5}	1.89×10^{-3}
errors in drawing surface	7.02×10^{-5}	1.41×10^{-4}
result	Figure 4.8(b)	Figure 4.8(a)

4.1.4 Optimizer, learning rate, epochs, and loss functions

As a choice of optimizer used when training the network, the Stochastic Gradient Descent (SGD) method was initially used. Although it had good performance when approximating 2D SDFs of basic primitives, it had poor performance in approximating

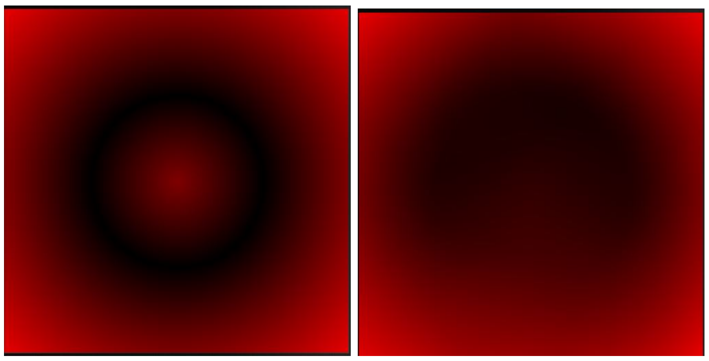


Figure 4.6. Rendering Result of CNN Trained with Dataset by Random Sampling

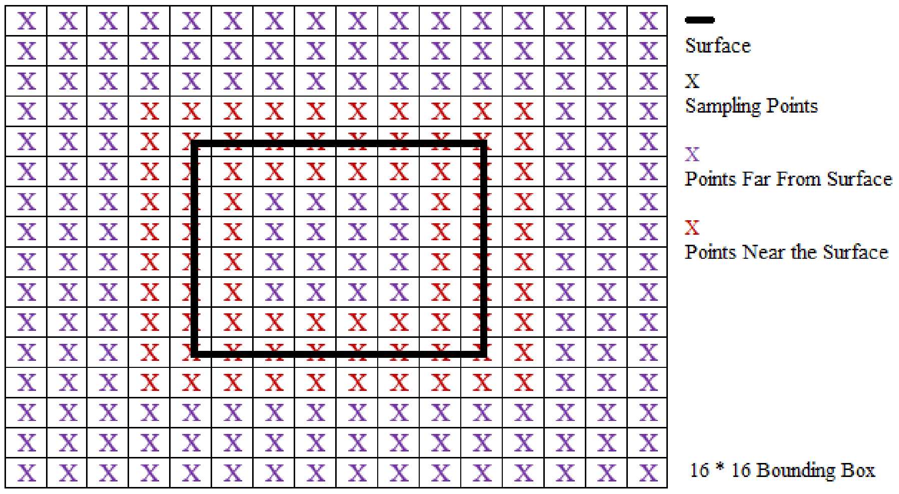


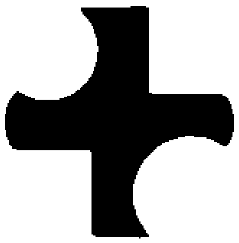
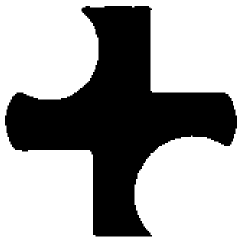
Figure 4.7. Sampling Approach

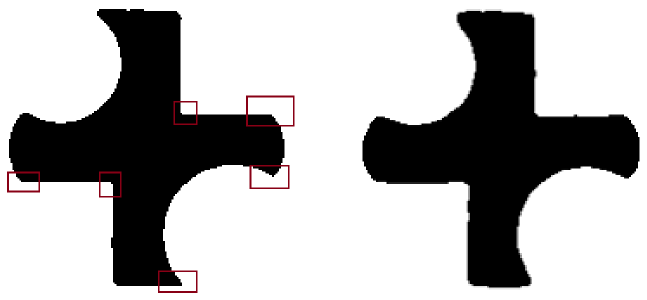
more complex SDFs. So we replaced it with Adaptive Moment Estimation (Adam) for the subsequent experiments.

For determining learning rates, we found that the learning rates ranging from 1.0×10^{-4} to 5.0×10^{-4} worked well for most experiments and did not have a large impact, as shown in Table 4.3.

In this research, one epoch means one training cycle to the dataset. With the increase in the number of training epochs, the fitting results are more accurate to the

Table 4.3. 2D SDFs Data Collection on GAN with Samples Near the Surface

#	Higher Learning Rate	Lower Learning Rate
learning rate	5.00×10^{-4}	1.00×10^{-4}
loss function	MSE and Errors by D-Model	MSE and Errors by D-Model
optimizer	Adam	Adam
overall errors(MSE)	2.56×10^{-3}	9.15×10^{-4}
errors in drawn pixels	1.64×10^{-4}	1.05×10^{-4}
result		



(a) Dataset by Narrow-band Surface Sampling (b) Dataset by Uniform Sampling

Figure 4.8. Rendering Results for ANNs with with Dataset by Different Sampling Methods

ground truth and have better details, as you can see in Figure 4.9. Also, according the collected errors in Figure 4.10(a), although the end of the curve is not decreasing as fast as the beginning, it is still becoming more and more regressive.

For loss functions, hybrid loss functions were used in experiments. The loss function for training CNNs was:

$$Error = 1.0 \times \frac{1}{n}\sum_{i=1}^{n}(\mathbf{P}_i - \mathbf{L}_i)^2 + 2.0 \times \frac{1}{n}\sum_{i=1}^{n}(\frac{\mathbf{P}_i^2}{-0.1} - \frac{\mathbf{L}_i^2}{-0.1})^2 \tag{4.1}$$

where **P** is a vector of n predictions and L is the vector of corresponding labels.

For GAN, the error calculated by the discriminative model needs to be added to the formula above:

$$Error = 1.0 \times \frac{1}{n}\sum_{i=1}^{n}(\mathbf{P}_i - \mathbf{L}_i)^2 + 2.0 \times \frac{1}{n}\sum_{i=1}^{n}(\frac{\mathbf{P}_i^2}{-0.1} - \frac{\mathbf{L}_i^2}{-0.1})^2 + 5.0 \times \frac{1}{n}\sum_{i=1}^{n}(\mathbf{PG}_i - \mathbf{LG}_i)^2 \tag{4.2}$$

where **PG** is a vector of n predictions by discriminative model and LG is a vector filled by of 1.0.

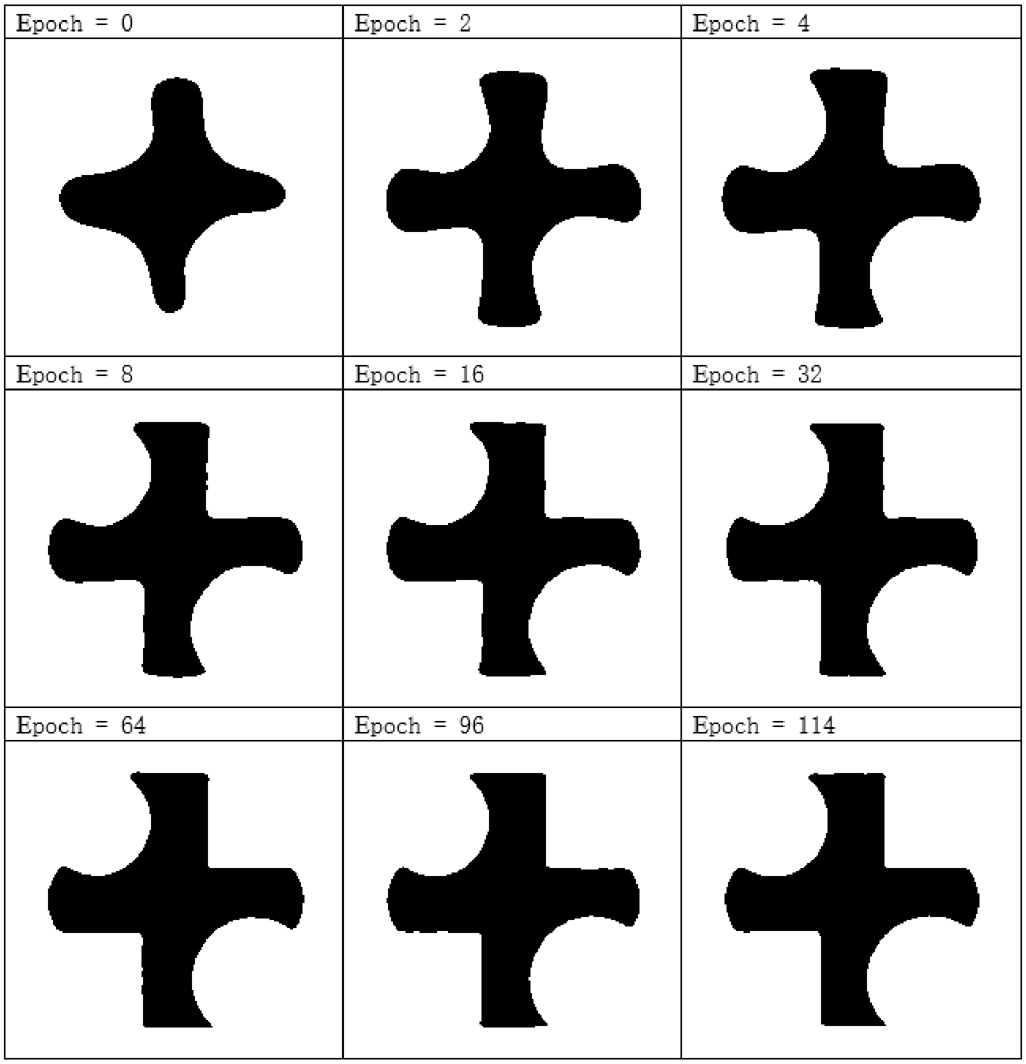


Figure 4.9. Rendering Results with Increasing Training Epochs

4.2 Rendering Method

The experiments used the following algorithm to look up the grid-based SDF and ANN based SDF.

In detail, the algorithm firstly checks if the point is inside of the bounding box, if so, the program will return the nearest distance to the bounding box plus a small value, if not, the program will return the nearest distance to the surface. A boolean value is also

Algorithm 4.1 Getting Distance from SDFs with Bounding Box

```
1: procedure LookUp
2:     $ro \leftarrow$ position of current point
3:     if $ro$ is inside of the bounding box then
4:         $distance \leftarrow$ nearest distance to the surface
5:         $flag \leftarrow$ True
6:     else
7:         $dist_0 \leftarrow$ nearest distance to the bounding box
8:         $dist_1 \leftarrow$ grid spacing of the bounding box
9:         $distance \leftarrow dist_0 + 0.5 \times dist_1$
10:        $flag \leftarrow$ False
11:    end ifreturn $flag, distance$
12: end procedure
```

returned by the program, which indicates if the distance is between surface or bounding box.

4.3 2D Signed Distance Functions

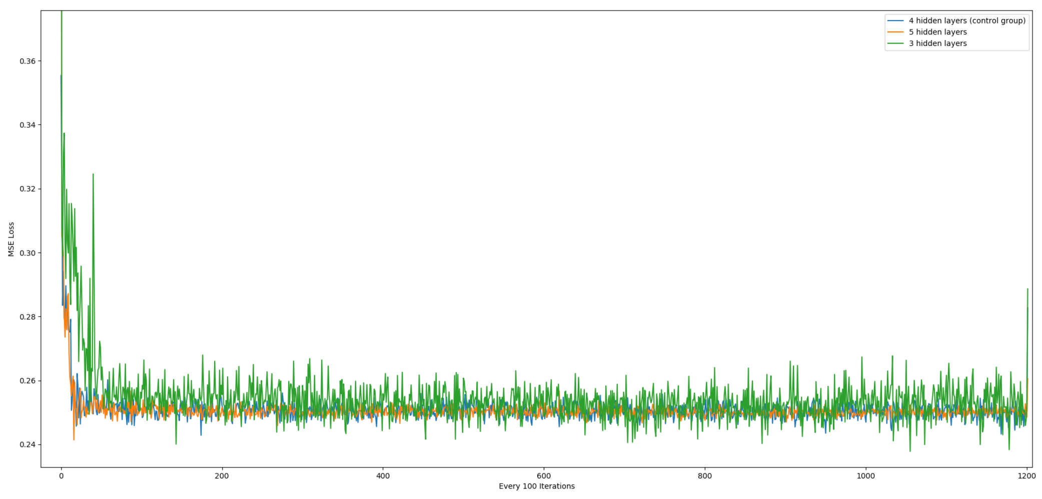
In this section, errors in training and rendering with different architectures of the ANNs for 2D SDFs are presented. Figure 4.9 showed how the rendering changed during training.

4.3.1 Training Errors

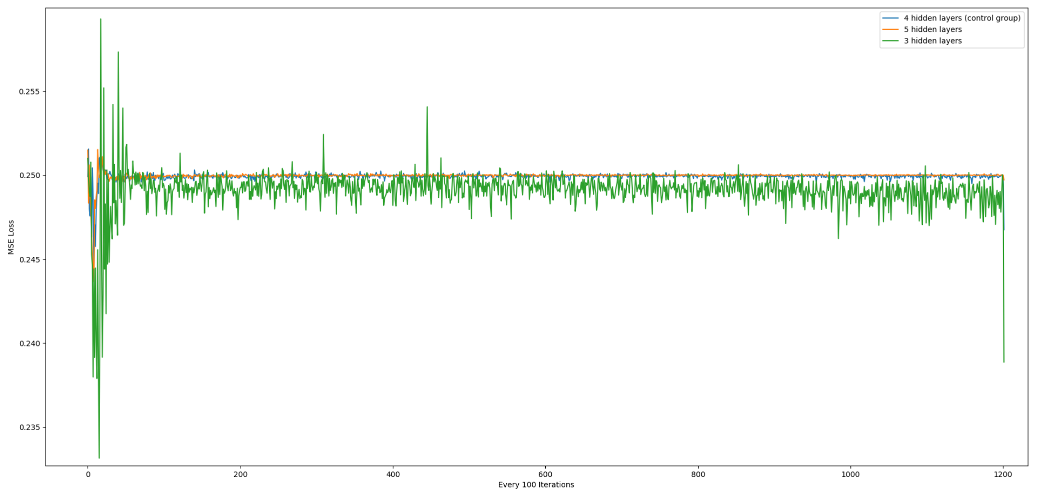
Figure 4.10 shows the errors that were collected during training. For both discriminative model and generative model, when the hidden layers of generative model increased, the regression happened faster with a smoother curve, though the difference between models with 4 hidden layers and models with 5 hidden layers was not significant.

4.3.2 Rendering Errors

Figure 4.11 shows the errors that were collected during testing, where Figure 4.11(b) means only errors in drawing pixels on the surface were measured and Figure



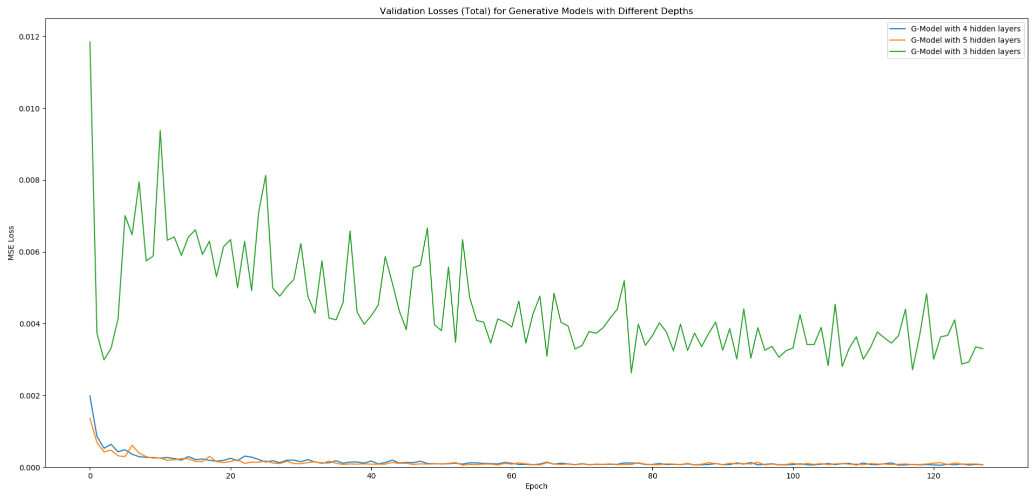
(a) Generative Model



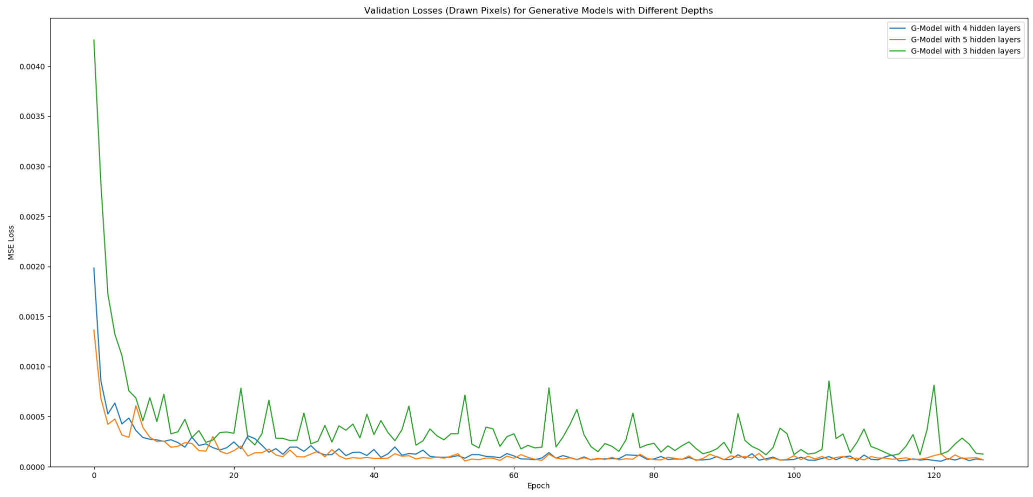
(b) Discriminative Model

Figure 4.10. Errors in Training GAN for 2D SDF

4.11(a) means errors in all pixels were measured. For total errors, the GAN with 3 hidden layers performed the worst, but there were no significant differences between the GAN with 4 hidden layers and the one with 5 hidden layers. For errors in drawing the surface, the GAN with 3 hidden layers performed better but still far from GANs with more layers, while the model with 4 hidden layers and the one with 5 still had similar performance.



(a) Errors in Overall Image



(b) Errors in Drawing the Surface

Figure 4.11. Errors in Rendering 2D SDFs Approximated by GAN

4.4 3D Signed Distance Functions

In this section, errors in training and rendering with different architectures of the ANNs for 2D SDFs are presented. Figure 4.12 shows the change in modeling 3D SDF by ANN after different number of training epochs.

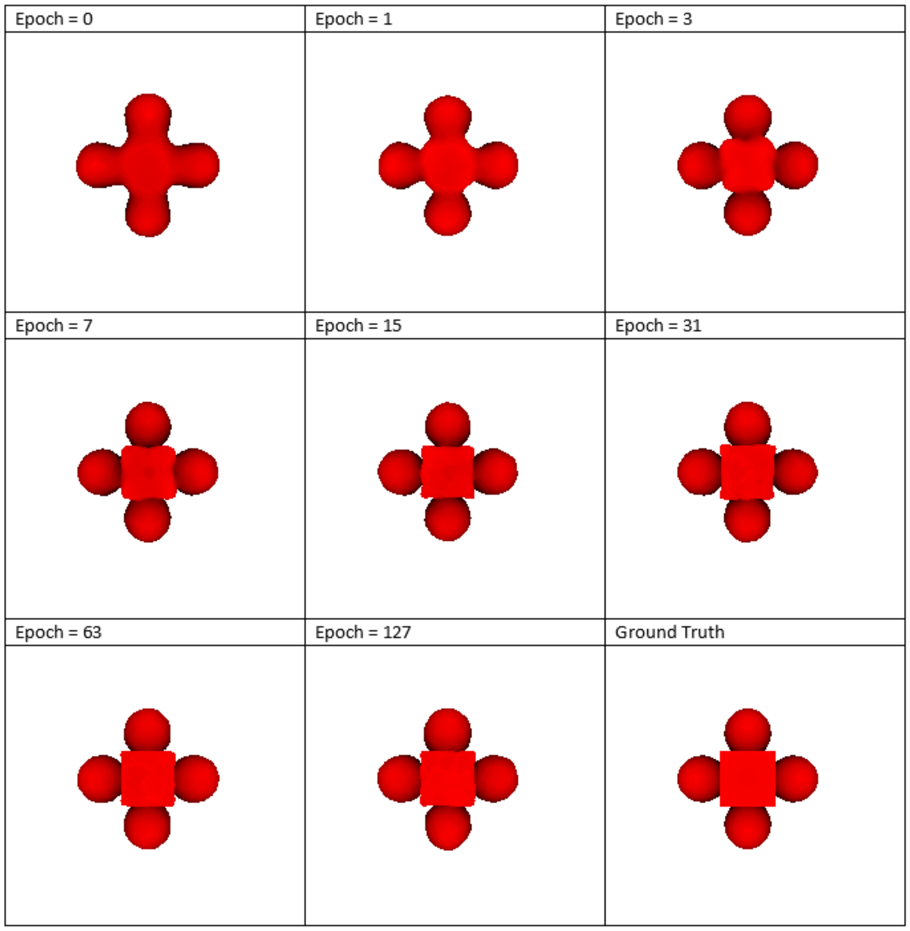
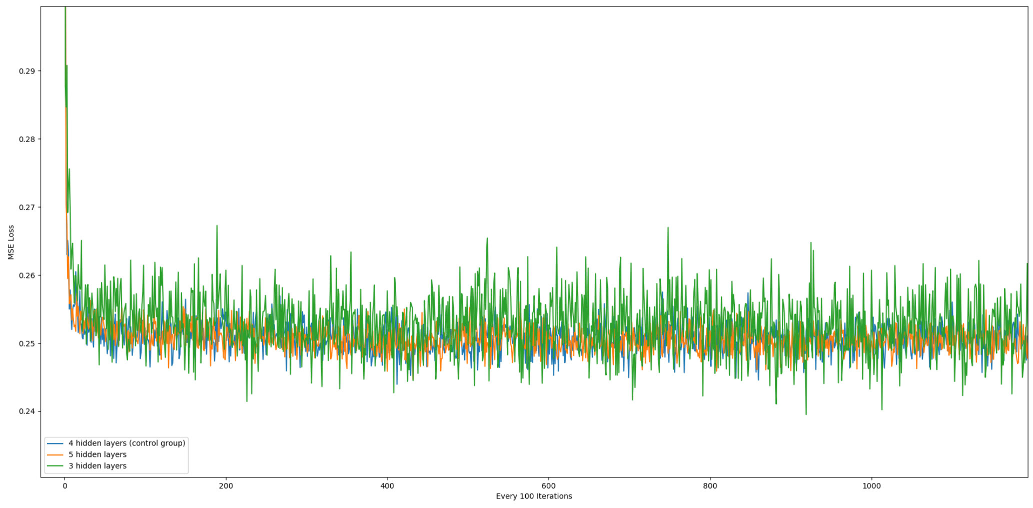


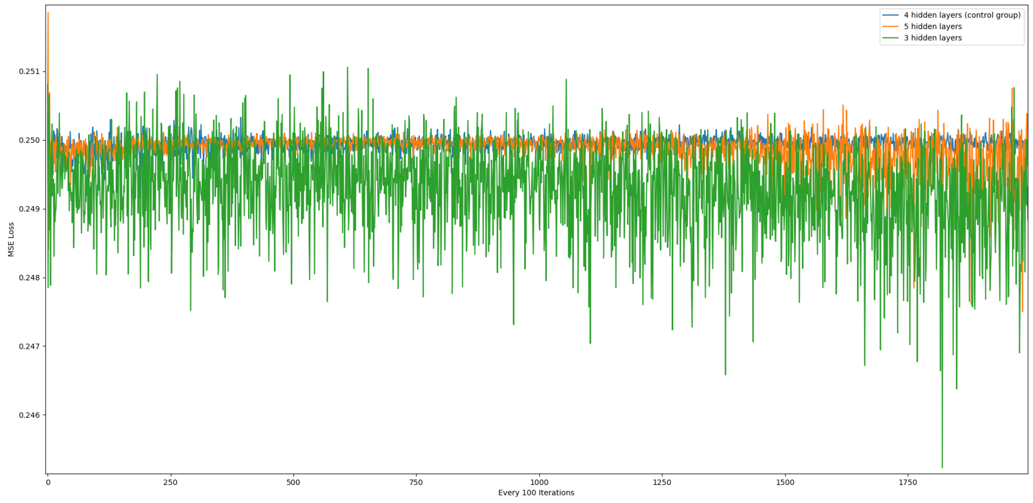
Figure 4.12. Rendering Results with Increasing Training Epochs

4.4.1 Training Errors

Figure 4.13 shows the training errors for ANNs in approximating 3D SDFs. For discriminative model, the overall curve was less regressive than the curve for training 2D SDFs, one of the main reasons is the dataset for 3D SDFs is more than 5 times larger than the dataset for 2D SDFs while the architectures of their ANNs are the same. From the figure, the ANN with 4 hidden layers performed the best and the one with 3 hidden layers performed the worst. The curve of errors in training generative model for 3D SDFs is similar to what for 2D SDFs, i.e. the deeper ANNs had lower errors.



(a) Generative Model



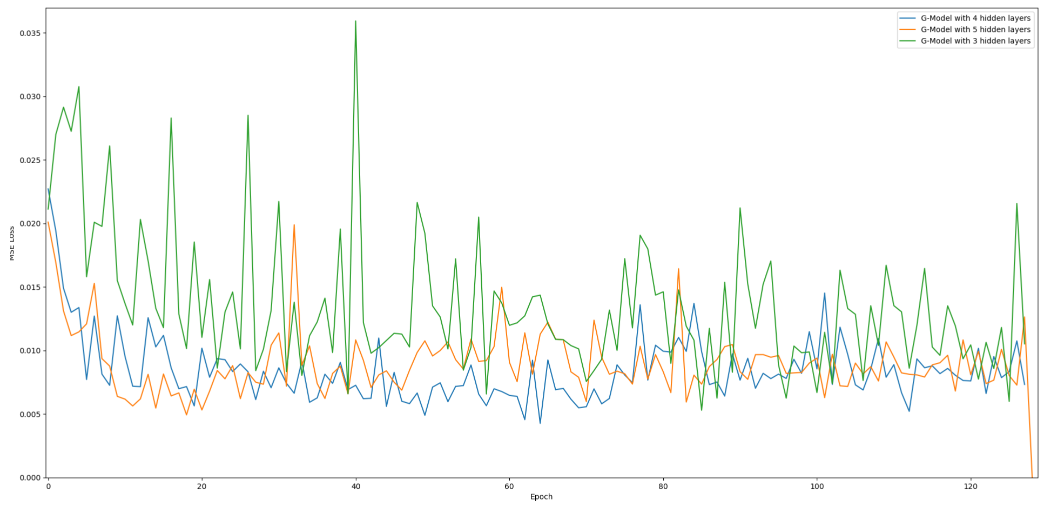
(b) Discriminative Model

Figure 4.13. Errors in Training GAN for 3D SDFs

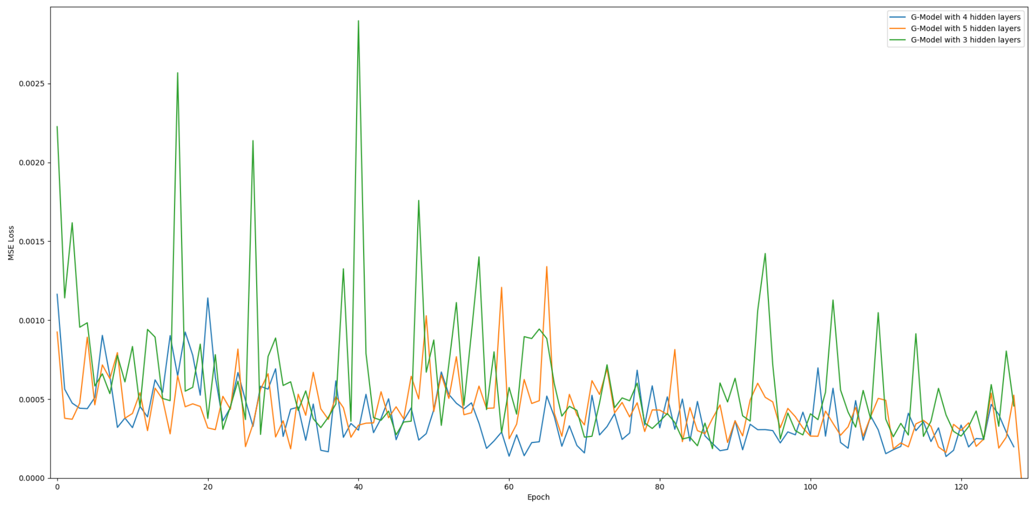
4.4.2 Testing Errors

Figure 4.14 shows the two different types of errors, the top image is about errors in fitting all voxel and the bottom one is errors in rendering the surface. The errors were measured by the distance between camera to the hit point on the surface by the MSE method, the distance is 0 if the ray hit nothing. Distance of ground truth was calculated by the analytical formulas that model the surface.

According to the Figure, the accuracies were worse than approximating 2D SDF or SDF to the triangle mesh, which is presented in the next section. One possible reason might be the surface is too sharp for the current architecture, which needs to be investigated in future experiments. Regardless of it, deeper *G*s still showed advantage over shallower architectures.



(a) Errors in All Pixels



(b) Errors in Drawing the Surface

Figure 4.14. Errors in Training GAN for 3D SDF

4.5 Triangle Mesh

In this section, errors in training and rendering with different architectures of the ANNs for mesh SDFs are presented. Besides, the section involves a comparison between ANN-based SDFs and grid-based SDFs, which is generated by level-set method with high dimensions. Figure 4.15 shows the rendering results with increasing training epochs.

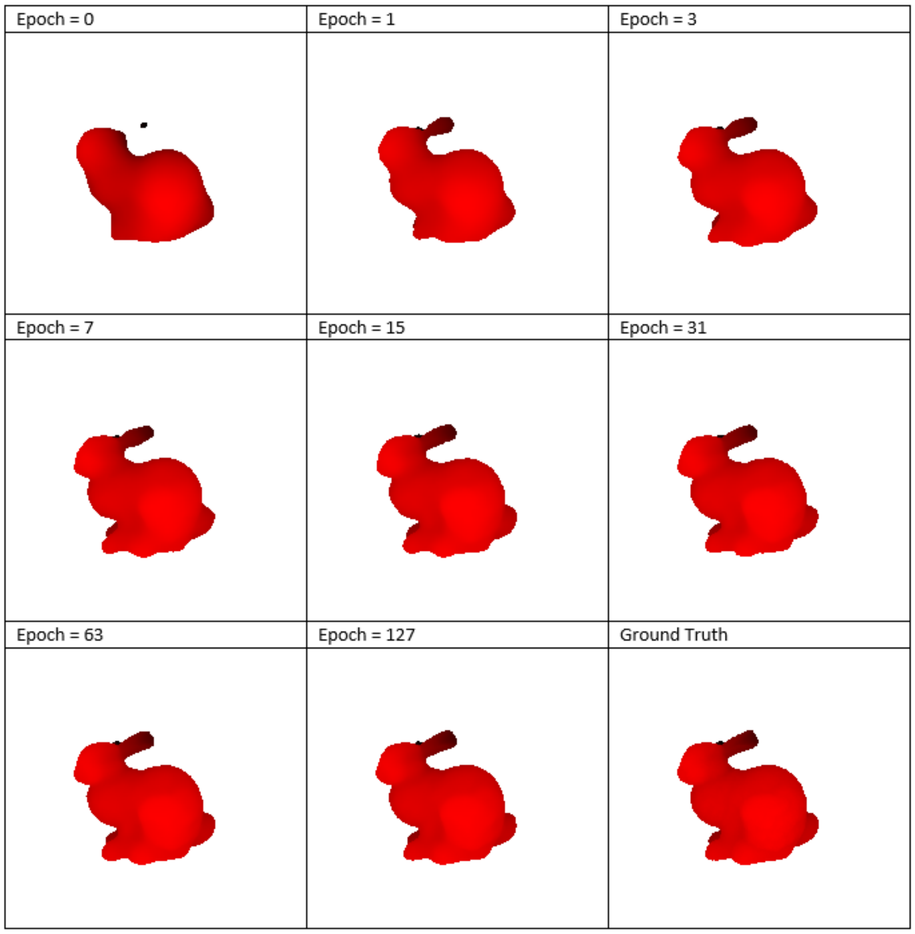
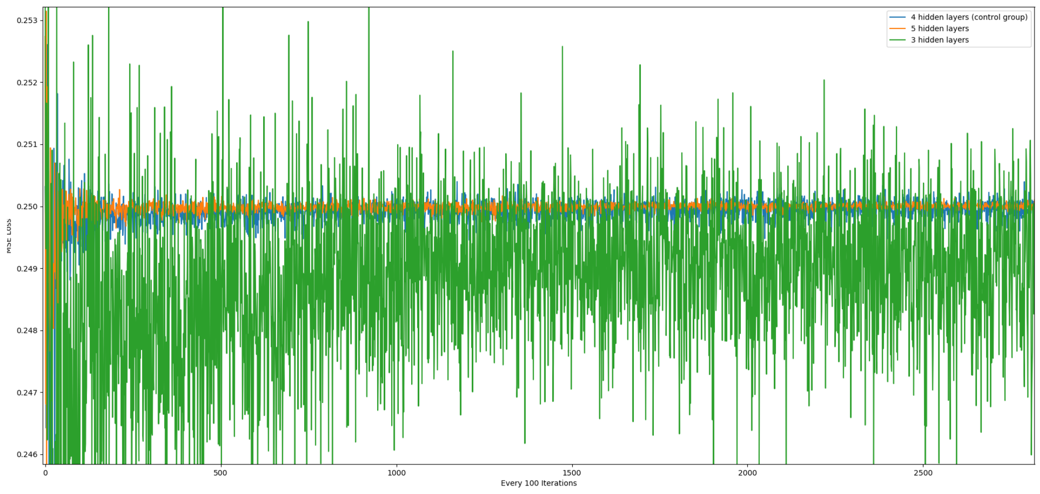


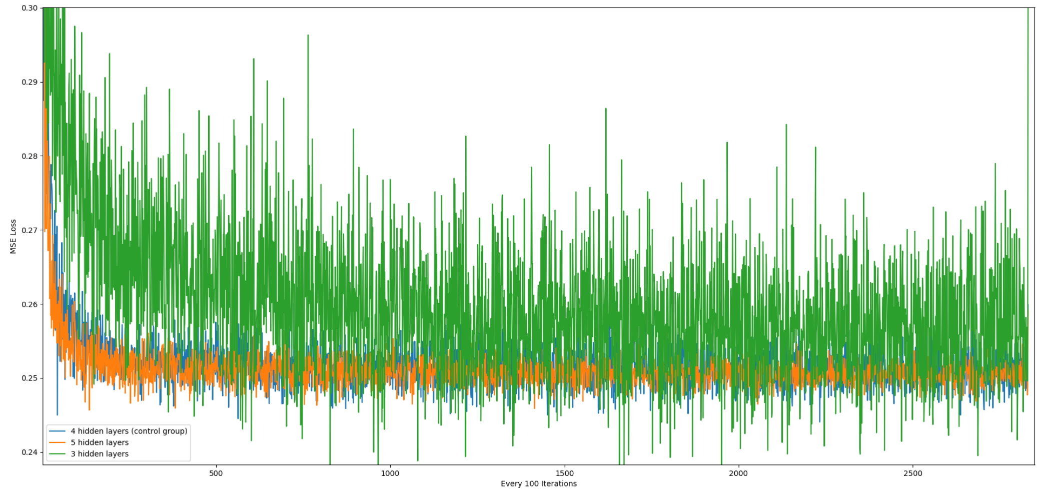
Figure 4.15. Rendering Results with Increasing Training Epochs

4.5.1 Training Errors

Figure 4.16(a) shows errors in training discriminative models, and Figure 4.16(b) represents the errors in training generative models. According to the figures, the training curve of D and G with the shallowest G model is very noisy compared to D with a deeper G model. For those with 4 hidden layers and 5 hidden layers, the difference is not significant in training G model, but the D model with a deeper G has a more regressive curve.



(a) Discriminative Model



(b) Generative Model

Figure 4.16. Errors in Training GAN for SDF to Triangle Mesh

4.5.2 Testing Errors

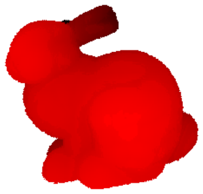
Figure 4.17 shows the errors in fitting all voxels and only in rendering pixels on the surface of the mesh. The errors were measured by the distance between camera to the hit point on the surface, the distance is 0 if the ray hit nothing. For efficiency, distances to ground truth were calculated by looking up a level-set SDF approximating the mesh, which has been demonstrated to be close to the real distance (Bærentzen & Aanæs, 2002; Osher & Sethian, 1988).

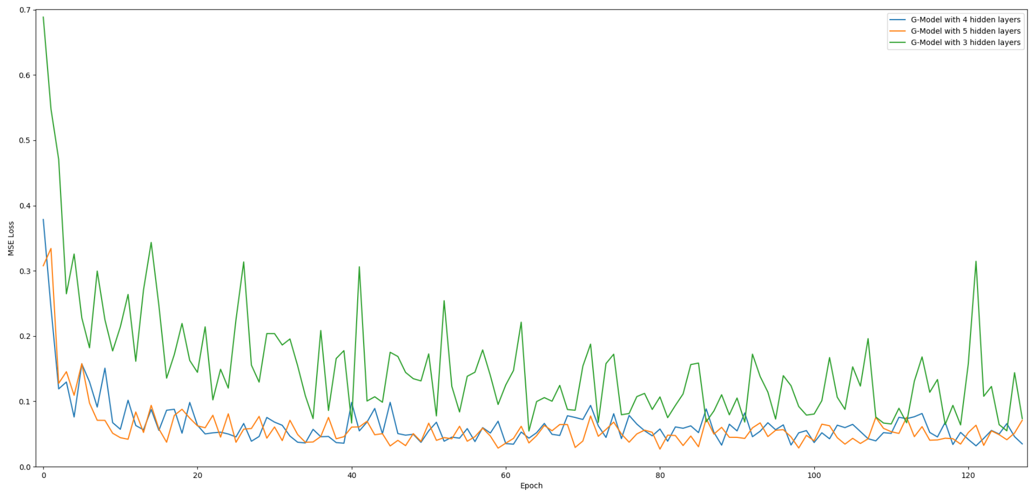
According to the Figure, the *G* with 3 hidden layers performed the worst while there is still no significant difference between the model with 4 hidden layers and the one with 5 hidden layers. And they do not show much decrease after the 40th epoch.

4.5.3 Grid-based Signed Distance Functions

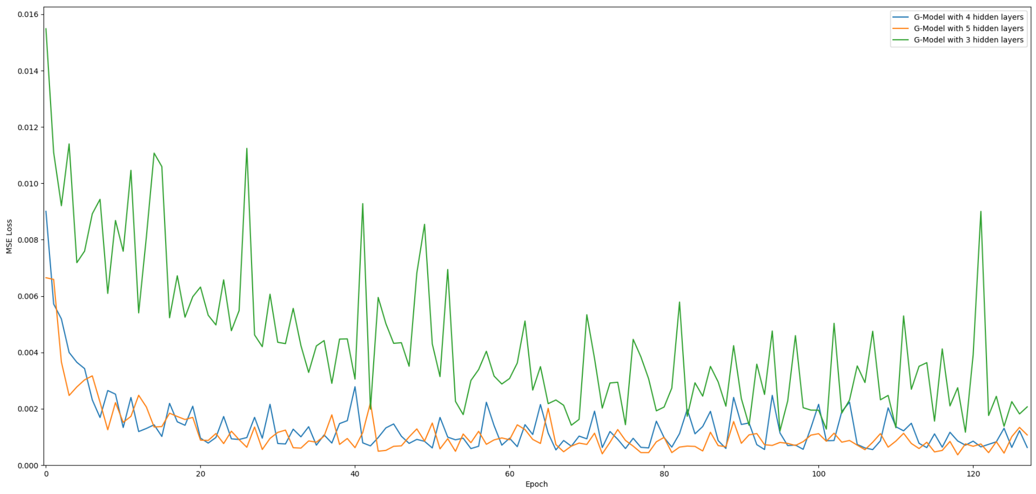
As shown in Table 4.4, the ANN successfully approximated the SDF to a triangle mesh while requiring a smaller memory size than the grid-based SDF using level-set method and even smaller than the triangle mesh itself. Because the rendering pipeline only needs generative model from the GAN to look up the distance, so only the size of *G* was included when measuring for the size of ANN-based SDF. The *G* is saved without compression in a binary format using a built-in function of PyTorch, which includes weights and biases for each layer of the ANN.

Table 4.4. Storage Size for Different Techniques

#	size	rendering result
triangle mesh	2.28MB	
grid-based SDF	33.9MB	
generative model	47KB	



(a) Errors in All Pixels



(b) Errors in Pixels on Surface

Figure 4.17. Errors in Training GAN for SDF to Triangle Mesh

CHAPTER 5. CONCLUSIONS

The goal of this research is to explore the potential of ANNs in learning SDFs and determine the optimal network architecture and training techniques. In this research, we implemented multiple types of ANNs with various depths of the generative model, which involved FCNN, CNN, and GAN. The final results narrowed the gap in using ANNs to learn 3D shapes while most previous work focused on 2D rendered images. Besides the computer graphics area, the framework of this research project could be applied to fit many other non-linear functions. Consequently, the main hypothesis of the research is demonstrated to be true, although future works remains, since this is a rapidly expanding field, and new ANN architectures and optimization technique can still be explored.

5.1 Summary of the Research

The research work explored the possibility of ANNs in learning SDFs by conducting three main experiments where each experiment was informed by results from the previous experiment.

The first experiment was using FCNN, CNN, and GAN to learn the SDF in 2D space. Analytic SDFs were used as the training data. The training dataset was sampled from the function using different techniques, including random sampling, uniform sampling, and narrow-band sampling. Based on these results, in the second experiment, a 3D shape rendered with similar formulas was used to train a GAN with a dataset obtained by narrow-band sampling. The increased size of the dataset made the training harder than the first experiment, although the GAN still was able to generate some plausible rendering results. Then in the third experiment, the same framework was applied to learn the SDF to a triangle mesh. First we generate a grid-based SDF by level-set method, and then use the dicretized SDF as the training data. In the experiment, the rendering result from GAN had a small error when compared to the grid-based SDF, and the storage size of the GAN is less than 0.1% of what the discretized SDF grid requires. In these experiments, errors in training and testing were collected and measured. The rendering results were also collected and presented in this thesis.

5.2 Future Work

The future work for this research includes:

- The training for 3D SDFs can be optimized further. As shown in last chapter, the training curve is sometimes noisy and over fitting.
- The narrow-band surface sampling can be optimized. It was useful in sampling from the 2D graph, but the training dataset became very large when applied to 3D.
- The potential of different architectures of the GAN should be researched in the future. This research has done a concise experiment on different widths and depths of the GAN, but there are more parameters to be explored to find which architecture is most efficient at learning SDFs.
- The current rendering framework is implemented on the CPU. The possibility to implement it on GPU can be explored.

REFERENCES

Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of the april 30–may 2, 1968, spring joint computer conference* (pp. 37–45).

Bærentzen, J. A., & Aanæs, H. (2002). Generating signed distance fields from triangle meshes. *Informatics and Mathematical Modeling, Technical University of Denmark, DTU*, *20*, 23.

Bærentzen, J. A., & Christensen, N. J. (2002). Volume sculpting using the level-set method. In *Shape modeling international, 2002. proceedings* (pp. 175–275).

Blanz, V., & Vetter, T. (1999). A morphable model for the synthesis of 3d faces. In *Proceedings of the 26th annual conference on computer graphics and interactive techniques* (pp. 187–194).

Blier, L. (2016). *A brief report of the heuritech deep learning meetup.* Retrieved from `https://heuritech.files.wordpress.com/2016/02/vgg16.png`

Carr, N. A., Hoberock, J., Crane, K., & Hart, J. C. (2006). Fast gpu ray tracing of dynamic meshes using geometry images. In *Proceedings of graphics interface 2006* (pp. 203–209).

Chrschn. (2007). *An example of a polygon mesh.* Retrieved from `https://commons.wikimedia.org/wiki/File:Dolphin_triangle_mesh.png`

Cohen, M. F., & Wallace, J. R. (2012). *Radiosity and realistic image synthesis.* Elsevier.

Csáji, B. C. (2001). Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, *24*, 48.

Denton, E. L., Chintala, S., Fergus, R., et al. (2015). Deep generative image models using a laplacian pyramid of adversarial networks. In *Advances in neural information processing systems* (pp. 1486–1494).

Donnelly, W. (2005). Per-pixel displacement mapping with distance functions. *GPU gems*, *2*(22), 3.

Enright, D., Marschner, S., & Fedkiw, R. (2002). Animation and rendering of complex water surfaces. *ACM Transactions on Graphics (TOG)*, *21*(3), 736–744.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672–2680).

Henrik. (2008). *Ray trace diagram.* Retrieved from `https://upload.wikimedia.org/wikipedia/commons/8/83/`

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., ... others (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, *29*(6), 82–97.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097–1105).

LeCun, Y., Bengio, Y., et al. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, *3361*(10), 1995.

LeCun, Y., et al. (2015). Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20.

Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A., Acosta, A., … others (2016). Photo-realistic single image super-resolution using a generative adversarial network. *arXiv preprint*.

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 3431–3440).

Malladi, R., Sethian, J. A., & Vemuri, B. C. (1995). Shape modeling with front propagation: A level set approach. *IEEE transactions on pattern analysis and machine intelligence*, *17*(2), 158–175.

Matsugu, M., Mori, K., Mitari, Y., & Kaneda, Y. (2003). Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Networks*, *16*(5-6), 555–559.

Maturana, D., & Scherer, S. (2015). Voxnet: A 3d convolutional neural network for real-time object recognition. In *Intelligent robots and systems (iros), 2015 ieee/rsj international conference on* (pp. 922–928).

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Medler, D. A. (1998). A brief history of connectionism. *Neural Computing Surveys*, *1*, 18–72.

Minsky, M., Papert, S. A., & Bottou, L. (2017). *Perceptrons: An introduction to computational geometry*. MIT press.

Möller, T., & Trumbore, B. (2005). Fast, minimum storage ray/triangle intersection. In *Acm siggraph 2005 courses* (p. 7).

Nalbach, O., Arabadzhiyska, E., Mehta, D., Seidel, H.-P., & Ritschel, T. (2017). Deep shading: Convolutional neural networks for screen space shading. In *Computer graphics forum* (Vol. 36, pp. 65–78).

Ng, R., Ramamoorthi, R., & Hanrahan, P. (2003). All-frequency shadows using non-linear wavelet lighting approximation. In *Acm transactions on graphics (tog)* (Vol. 22, pp. 376–381).

Osher, S., & Sethian, J. A. (1988). Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations. *Journal of computational physics*, *79*(1), 12–49.

Quilez, I. (2013). *Modeling with distance functions.* Retrieved from `http://iquilezles.org/www/articles/distfunctions/distfunctions.htm`

Ren, P., Dong, Y., Lin, S., Tong, X., & Guo, B. (2015). Image based relighting using neural networks. *ACM Transactions on Graphics (TOG)*, *34*(4), 111. Retrieved from `http://doi.acm.org/10.1145/2766899`

Ren, P., Wang, J., Gong, M., Lin, S., Tong, X., & Guo, B. (2013). Global illumination with radiance regression functions. *ACM Transactions on Graphics (TOG)*, *32*(4), 130. Retrieved from `http://doi.acm.org/10.1145/2461912.2462009`

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533.

Rumelhart, D. E., & McClelland, J. L. (1988). *Explorations in the microstructure of cognition (volume 1: Foundations).* MIT press.

Sato, S., Morita, T., Dobashi, Y., & Yamamoto, T. (2012). A data-driven approach for synthesizing high-resolution animation of fire. In *Proceedings of the digital production symposium* (pp. 37–42).

Séquin, C. H. (1987). Procedural spline interpolation in unicubix.

Shen, Y., He, X., Gao, J., Deng, L., & Mesnil, G. (2014). Learning semantic representations using convolutional neural networks for web search. In *Proceedings of the 23rd international conference on world wide web* (pp. 373–374).

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... others (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484–489.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Stergiou, C., & Siganos, D. (n.d.). *Neural networks.* Retrieved from `https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.simple_neuron.jpg`

Svozil, D., Kvasnicka, V., & Pospichal, J. (1997). Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, *39*(1), 43–62.

Thomas, M. M., & Forbes, A. G. (2017). Deep illumination: Approximating dynamic global illumination with generative adversarial network. *arXiv preprint arXiv:1710.09834*.

Tompson, J., Schlachter, K., Sprechmann, P., & Perlin, K. (2016). Accelerating eulerian fluid simulation with convolutional networks. *arXiv preprint arXiv:1607.03597*.

Ulyanov, D., Lebedev, V., Vedaldi, A., & Lempitsky, V. S. (2016). Texture networks: Feed-forward synthesis of textures and stylized images. In *Icml* (pp. 1349–1357).

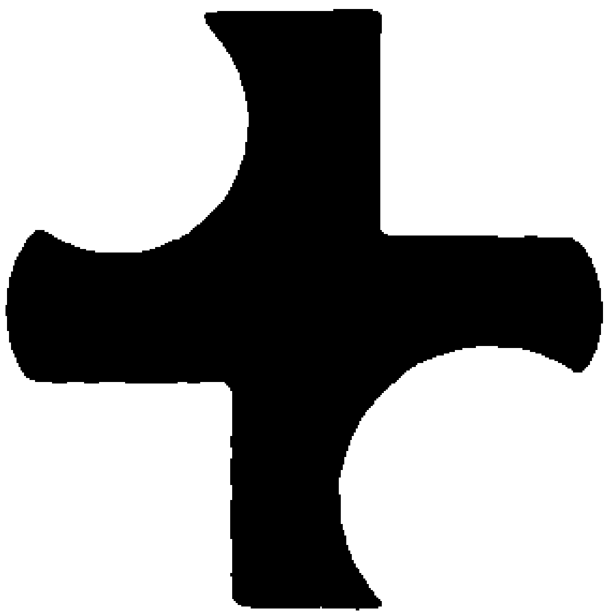
Whitaker, R., Breen, D., Museth, K., & Soni, N. (2001). Segmentation of biological volume datasets using a level-set framework. In *Volume graphics 2001* (pp. 249–263).

Zhao, H., Gallo, O., Frosio, I., & Kautz, J. (2015). Is l2 a good loss function for neural networks for image processing? *ArXiv e-prints*, *1511*.

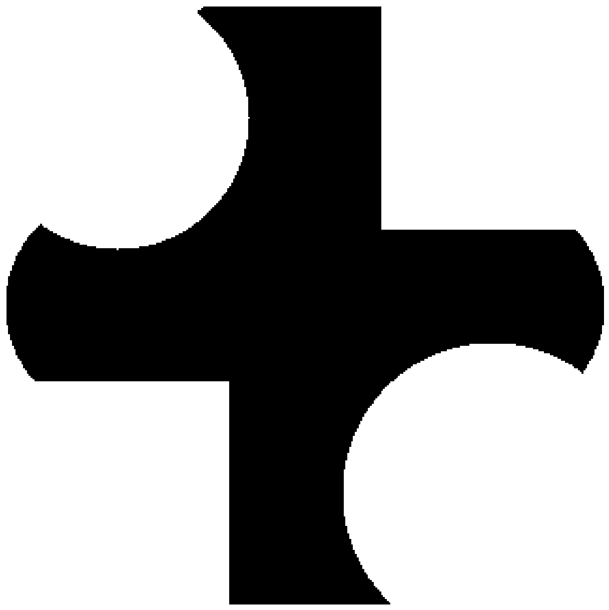
Zhou, K., Ren, Z., Lin, S., Bao, H., Guo, B., & Shum, H.-Y. (2008). Real-time smoke rendering using compensated ray marching. In *Acm transactions on graphics (tog)* (Vol. 27, p. 36).

APPENDIX A. RENDERING RESULTS WITH HIGH RESOLUTION

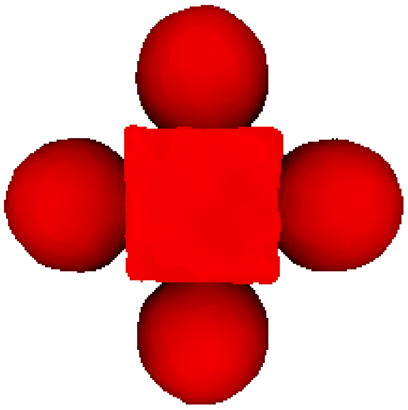
Rendering results as well as collected errors in the thesis are using a camera with a resolution of 200×200 pixels. The following figures show the results with higher resolution. The model does not need to be retrained when it is applied to a new camera with different resolutions, transform, or field of view.



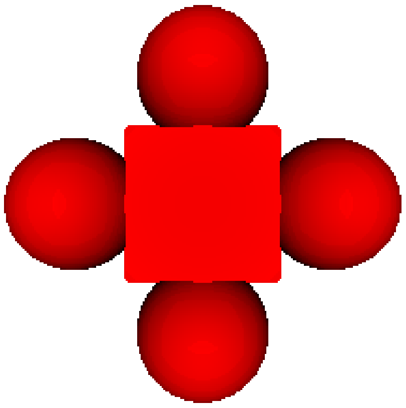
(a) ANN



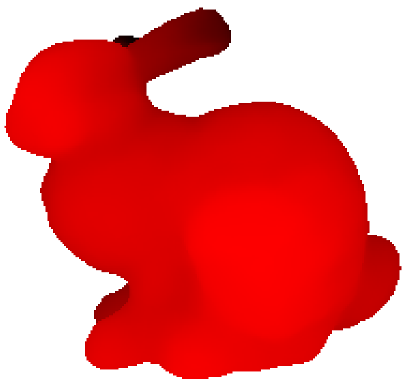
(b) Ground Truth



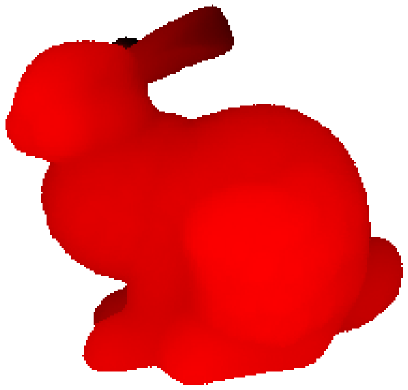
(c) ANN



(d) Ground Truth

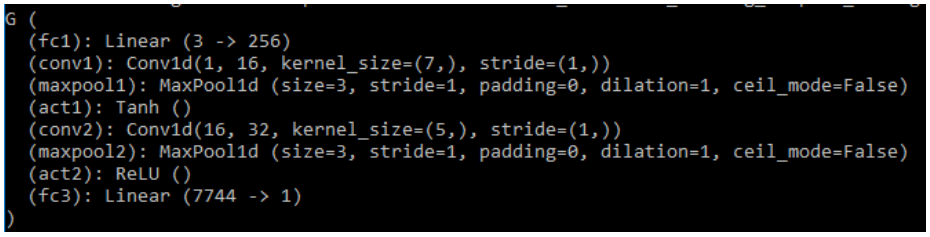


(e) ANN



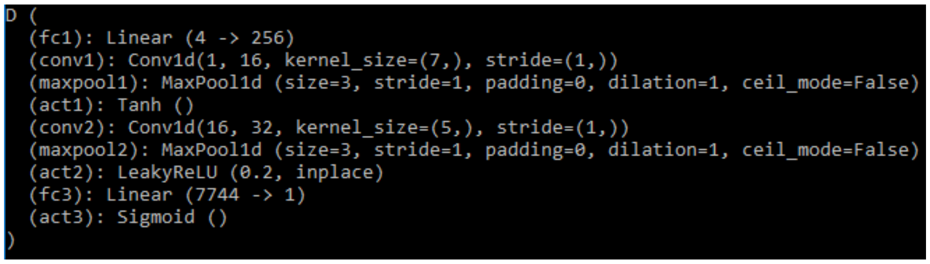
(f) Ground Truth

APPENDIX B. ARCHITECTURES OF GAN



```
G (
  (fc1): Linear (3 -> 256)
  (conv1): Conv1d(1, 16, kernel_size=(7,), stride=(1,))
  (maxpool1): MaxPool1d (size=3, stride=1, padding=0, dilation=1, ceil_mode=False)
  (act1): Tanh ()
  (conv2): Conv1d(16, 32, kernel_size=(5,), stride=(1,))
  (maxpool2): MaxPool1d (size=3, stride=1, padding=0, dilation=1, ceil_mode=False)
  (act2): ReLU ()
  (fc3): Linear (7744 -> 1)
)
```

(a) Generative Model



```
D (
  (fc1): Linear (4 -> 256)
  (conv1): Conv1d(1, 16, kernel_size=(7,), stride=(1,))
  (maxpool1): MaxPool1d (size=3, stride=1, padding=0, dilation=1, ceil_mode=False)
  (act1): Tanh ()
  (conv2): Conv1d(16, 32, kernel_size=(5,), stride=(1,))
  (maxpool2): MaxPool1d (size=3, stride=1, padding=0, dilation=1, ceil_mode=False)
  (act2): LeakyReLU (0.2, inplace)
  (fc3): Linear (7744 -> 1)
  (act3): Sigmoid ()
)
```

(b) Discriminative Model

APPENDIX C. CAMERA, MESH, AND GRID-BASED SDF INFORMATION

#	Information
filename	bunny.obj
encoding	*ANSI*
size	2.4 MB
number of vertices	$34,836$
number of triangles	$69,664$
number of normals	N/A
texture coordinates	N/A
vertex color	N/A

#	Information
filename	bunny.sdf
encoding	*ANSI*
size	33.9 MB
field dimensions	$165 \times 164 \times 130$
grid origin	$[-2.07122 \quad -2.05417 \quad -1.6334]$
grid spacing	0.025

#	Information
filename	*camera.py*
type	Pinhole Camera
resolution	200×200
camera origin	$[0.0 \quad 0.0 \quad 0.0]$
horizontal field of view	90°