FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Image-Space Metaballs Using Deep Learning

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Robert Horvath, BSc
Matrikelnummer 01025519

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Projektass. Dipl.-Ing. Dr.techn. Ivan Viola
Mitwirkung: Projektass. Ing. Dr.techn. Peter Mindek

Wien, 24. Juli 2019

Robert Horvath	Ivan Viola

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Image-Space Metaballs Using Deep Learning

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Robert Horvath, BSc
Registration Number 01025519

to the Faculty of Informatics

at the TU Wien

Advisor: Projektass. Dipl.-Ing. Dr.techn. Ivan Viola
Assistance: Projektass. Ing. Dr.techn. Peter Mindek

Vienna, 24th July, 2019

Robert Horvath	Ivan Viola

Erklärung zur Verfassung der Arbeit

Robert Horvath, BSc
Mellingerstrasse 130B, 5400 Baden, Schweiz

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. Juli 2019

Robert Horvath

Kurzfassung

Metaballs sind eine Klasse von impliziten Flächen die verwendet werden um organisch aussehende Formen und Flüssigkeiten zu modellieren. Exakte Darstellung von drei-dimensionalen Metaballs wird typischerweise durch Ray-casting erzielt, das jedoch rechenaufwendig ist und nicht für Echtzeitanwendungen geeignet ist. Deshalb wurden verschiedene annähernde Rendering Methoden entwickelt.

In dieser Arbeit werden die Grundlagen von Metaballs und Neuralen Netzwerken behandelt, und ein neuer Ansatz um Metaballs durch Deep Learning zu rendern der performant genug für Echtzeitanwendungen ist präsentiert. Dieses System verwendet einen Bild-zu-Bild-Übersetzungs Ansatz. Dafür werden die Metaballs zuerst in einer stark vereinfachten Form in ein Bild gerendert. Dieses Bild dient danach als Input für ein Neurales Netzwerk das Tiefe, Normalen, und Grundfarben-Puffer als Output produziert die anschließend mit einem deferred shading Renderer in ein finales Bild kombiniert werden können.

Abstract

Metaballs are a type of implicit surface that are used to model organic-looking shapes and fluids. Accurate rendering of three-dimensional metaballs is typically done using ray-casting, which is computationally expensive and not suitable for real-time applications, therefore different approximate methods for rendering metaballs have been developed.

In this thesis, the foundations of metaballs and neural networks are discussed, and a new approach to rendering metaballs using Deep Learning that is fast enough for use in real-time applications is presented. The system uses an image-to-image translation approach. For that, first the metaballs are rendered using a very simplified representation to an image. This image is then used as input to a neural network that outputs a depth, normal and base color buffer that can be combined using a deferred shading renderer to produce a final image.

Contents

CHAPTER 1

Introduction

1.1 Motivation

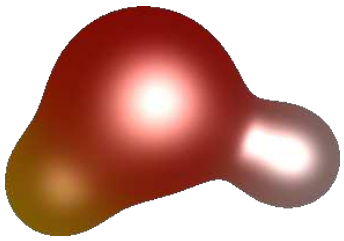


Figure 1.1: Metaballs.

Since the dawn of computer graphics people have been striving to find a way to represent the world around them as accurately as possible. However, different kinds of objects, like manmade and naturally occurring objects, lend themselves to different representations. Meshes often provide a great way to describe manmade objects. However, meshes are not ideal for describing every type of object such as organic-looking structures or structures that change topologically over time. These natural or ever-changing objects are often easier to describe with analytical formulas than meshes, and these have the added advantage of using less storage than equivalent meshes.

One approach to render organic shapes such as fluids or molecular models are metaballs. Metaballs have a long history in the computer graphics field. They were first introduced by Jim Blinn in the early 1980s as a proposed improvement over the traditional stick-and-ball model of rendering molecules and went on to be applied to create other organic looking shapes, for example liquid drops [XW16]. They have come to be known informally as "blobby objects", and thanks to faster hardware and algorithmic improvements, they have come some way since their introduction.

As beneficial as metaballs are for their improved ability to represent organic structures, rendering them is still costly compared to rendering comparable objects made from meshes. That is because the formulas used to describe metaballs are often not trivial to evaluate, and they have to be performed many times in the classic rendering approaches such as ray-marching. As the topology of naturally occurring objects may change over time, the surface requires constant re-evaluation. Tessellation of the surface (wherein the surface of an object is converted to a mesh) using an algorithm such as marching cubes is thus found to be inefficient as the conversion process has to be performed repeatedly.

Therefore, approximate approaches to render metaballs are often used in real-time applications. These apparent, or "fake" metaballs look similar to actual metaballs, and importantly behave similarly, but only approximate the exact shape that metaballs would produce. Many such approximate rendering techniques for metaballs have been proposed and are discussed in Chapter 2.1.

In recent years, deep learning has had great success in the field of computer graphics and image generation. Therefore, we ask the question if deep learning technology could provide a new, fast way to render apparent metaballs by learning to transform a description of metaballs into an image.

1.2 Problem Statement

Rendering metaballs accurately is computationally expensive and the number of metaballs that can be rendered with real time frame rates is limited. Attempts are still being made to try and find faster ways. The goal of this thesis is to produce a renderer using deep learning that is fast enough for real time applications, trading some accuracy for performance.

1.3 Approach

The approach is based on previous work of Isola et al. [IZZE17] in image-to-image translation using deep neural networks. We construct an input image describing the scene of metaballs and train a network to produce a final output image from it.

1.4 Contribution

In this thesis, a pipeline was created to enable the training of a network for the task of rendering metaballs. A renderer was also implemented to render metaballs using the trained network.

1.5 Structure of the work

At first, the state of the art in both metaball rendering and neural networks in the area of image processing is examined in Chapter 2.
Afterwards, in Chapter 3, the basics and background of the basic building blocks that are used in this thesis are discussed. Implicit surfaces, their mathematical definition, and how they are traditionally rendered are explained first, before going into more detail about metaballs, a special case of implicit surfaces. In the second half of that chapter we will discuss artificial neural networks, with focus on certain network types that have been proven successful for image-related tasks, namely CNNs and GANs. At the end, deferred shading, a rendering technique that guided the type of output we trained our network on is explained.
Building on the information given in Chapter 3, the next chapter describes the planned approach to generate images of approximate metaballs with the help of neural networks. First we discuss the characteristics of the training dataset and how it is generated. Then, the architecture of our network is shown. At the end, we look at how to use the network in a larger context and how it can be embedded in an existing renderer or interactive application.
The actual implementation of the system is presented in Chapter 5. We start with details on how the dataset is generated, then the implementation of the neural network itself follows. An interactive demo application has been developed as part of this thesis, and its implementation is also presented in that chapter.
The results of the work are presented in Chapter 6. Examples of input/output pairs are shown, and measurements of performance of the different stages of the pipeline are presented.
The results are then evaluated and placed in the context of related research in Chapter 7, and possible improvements on this method are hypothesised.
Finally, Chapter 8 presents a summary of this thesis.

CHAPTER 2

State of the Art

2.1 Metaballs

The history of metaballs started in the 1980s with Jim Blinn [Bli82]. Since then, a lot of advances in computer graphics and rendering have been made in general. But in this section we take a look at some particular techniques that have been presented for render metaballs.

Because solving the field equations for metaballs analytically is hard, ray-marching and marching cubes algorithms, that rely on evaluating the field equation only at discrete sampling points, were among the first and simplest methods for displaying metaballs. These algorithms require many steps for rendering though, so some focused on finding ways to calculate the ray-isosurface intersection faster. An example of such a method is described in Nishita et al.'s 1994 paper [NN94]. They used a 6-degree polynomial field function with finite support, and first found intervals of intersections between the ray and the effective spheres of the metaballs. If an interval only crosses the effective sphere of a single metaball, then the intersection can be found using a sphere-ray intersection test. Otherwise, the field function along the ray in the interval is approximated using Bézier curves. They use Bézier clipping, a technique to find the intersection between Bézier curves, also introduced by Nishita, to find the roots of the approximated field function, and thus the isosurface intersection point.

This approach was later refined by Kanmori et al. in 2008 [KSN08] by making use of the GPU to parallelize the computation of the intersection intervals. They use a technique called depth peeling to efficiently find intersection intervals, on which they perform Bézier clipping afterwards. Depth peeling works by first rendering the effective spheres of all metaballs, which gives the depth of the first ray-sphere intersection for each visible effective sphere along each ray. In successive rendering passes, only fragments that have a higher depth (are farther away) are rendered by using the previous rendering pass as a

kind of depth mask. This gives the next intersections, and thereby allows the intersection intervals to be retrieved.

Others took a different approach by relaxing different properties of metaballs some more. Adams et al. 2006 [ADL06] used splatting to draw spheres in place of metaballs and blend certain attributes, such as color and normals, in the overlapping regions to obtain smoothly blended surfaces, taking special care that depth-edges are preserved. This simulates the smooth blending achieved with metaballs, but does not result in the outline of the shape being blended smoothly. It still achieves good results for dense metaball surfaces.

Accelerating ray-casting with specialized data structures is another approach that has been followed by various researchers. Gourmel et al. 2010 [GPP+10] use a fitted bounding volume hierarchy together with a kd-tree as acceleration data structure in order to reduce the number of evaluations needed along the ray during ray casting. Bolla 2010 [Bol10] uses a perspective grid that stores metaballs intersecting with each voxel of the grid as a support data structure for accelerated ray-casting.

In the paper by Szécsi et al. 2012 [SI12], an algorithm that builds a list per pixel of the relevant metaballs, then incrementally constructs a piecewise polynomial approximation of summed metaball densities along the rays, and then uses them to find ray intersections with the isosurface is presented.

Approaches based on the marching cubes algorithm have also seen advances. Krone et al. 2012 [KSES12] present a method to compute a volumetric density map from metaballs with a Gaussian density function, and use a GPU-accelerated marching cubes algorithm to extract the isosurface from the density map.

Müller et al. 2007 [MGE07] presented two image space approaches to rendering metaballs. In their first approach they pre-compute a vicinity texture, in which they store, for each metaball particle, the neighbors of the metaball whose effective spheres overlap with its own. These are the metaballs that can contribute to the accumulated density value within its effective sphere. When rendering a metaball, its neighbors are looked up in the texture. If it has no neighbors, then the metaball can simply be drawn as a sphere. Otherwise, the surface is determined by raycasting in the fragment shader, where the field function is sampled only within the bounds of the effective spheres of the metaball and its neighbors. Their second approach uses a different kind of helper textures. In the texture the minimum and maximum distance of each fragment is stored, initialized by ray-casting the front and back sides of each metaballs effective sphere. In successive rendering passes, the field values at these distances is computed and the texture updated to move closer to the actual isosurface.

2.2 Neural Networks

Yann LeCun is considered a founding father of convolutional neural networks (CNNs) [LBB+98]. These kinds of networks are especially designed to deal with 2D shapes and images. Early on, they proved successful in character and digit recognition tasks. Since then, neural networks for operating on image data have seen dramatic improvements.

Greater insight into the operations of CNNs was enabled through a paper in 2009 that presented a way to visualize the higher level features that a network learns and uses to classify objects [EBCV09].

To help evaluate the progress in image recognition tasks, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has been created. It is a benchmark in object category classification and detection on hundreds of object categories and millions of images [RDS+14]. ImageNet [DDS+09] itself is a large scale hierarchical image database, containing over 14 million annotated photographs at this point, categorized into more than 21 thousand classes. The ILSVRC was held annually between 2010 and 2017 using subsets of the ImageNet data of approximately 1 million images and 1000 different classes. For each challenge, a labeled training dataset was released. Another dataset of unlabeled images was also released, for which classes had to be determined. Results were sent to a server for evaluation.

The participants of the challenge have pushed back the frontier of computer vision, deep learning and artificial intelligence each year. Success has primarily been achieved by deep convolutional neural networks on GPU hardware. The winner of the 2010 challenge, AlexNet [KSH12] achieved a classification error rate of 17%. AlexNet was also the winner of the 2012 competition, with an error rate of 15.3%. Over the coming years, classification error rates fell to just a few percent. In 2014, GoogLeNet [SLJ+15] achieved a classification error rate of 6.67%, and in the following year ResNet [HZRS16] achieved an error rate of just 3.57% - beating human performance at the classification task rated at 5%. The following years the error rate was improved slightly, using ensembles of models from the previous years winners.

Even though deep neural networks got better and better at object recognition, a paper in 2013 demonstrated that by adding some noise to images, that is basically invisible to the human eye, neural networks can be fooled into not being able to recognize the object or even into thinking it falls into an entirely different class [SZS+13].

Convolutional Neural Networks have found great success in other areas than object recognition as well. Amongst other uses, they have also been used to generate images. One basic architecture in this field is the auto-encoder, which through a series of convolutional layers learns to extract a representation of the important abstract features of the input image distribution, then through a series of deconvolutional layers also learns a way to reconstruct the original input from the learned representation.

Various types of auto-encoder networks exist. One class of auto-encoder is the de-noising auto-encoders, which has been used to create networks that clean up and remove noise

from their input images. They've also been used for image inpainting, a technique in which areas of an input image are deleted and the network fills in the holes with appropriate image data from the surrounding context. This can be used for removing obstructions from an image or general image retouching [XXC12].

Other interesting image processing applications using deep convolutional neural networks have been developed, such as a network for colorizing black and white photos [CYS15], or recreating an input image in the style of another input image, a process known as style transfer [GEB16]. Nalback et al. 2017 [NAM+17] used deep convolutional neural networks to learn and apply a number of screen-space shading effects to images, such as ambient occlusion, depth-of-field, motion blur, and more.

A breakthrough in image generation networks was achieved by Generative Adversarial Networks (GANs), introduced in 2014 [GPAM+14]. This architecture learns an input image distribution and is able to create completely new samples that fit the distribution and have a realistic look. GANs will be described in more detail later in this thesis in Section 3.3.5.

Besides new sample generation, GANs have also provided the basis for other image tasks like super resolution. In the paper by Ledig et al. 2017 [LTH+17] a GAN is used for 4x upscaling of photo-realistic images while retaining high quality.

With the advance of image generation, GANs and other techniques have raised concerns about security and privacy in recent years [CC18]. It is becoming increasingly easy to automatically replace faces in images and videos with the face of another person, for example. This has led to a number of "deep fakes" appearing on the internet, in which eg. celebrities might have their face swapped onto pornographic material. Other recent innovations have demonstrated taking a video of a character talking and modifying the video to match the lip movement to a new track of speech audio [SSKS17]. A recent publication by Fried et al. takes this further and demonstrates a system to edit a video of a character talking by simply editing the transcript of what the character says. The system synthesizes audio for the new text and replaces the lower face region to match the new words, resulting in a high quality fake video [FTF+19].

CHAPTER 3

Background

3.1 Implicit surfaces

3.1.1 Surfaces

Surfaces, in the mathematical sense, are a continuous set of points that have length and breadth but no thickness. They are two-dimensional shapes that exist in three dimensional space.

One way to store the representation of a surface is to explicitly store vertices located on the surface and the connections between them, describing a polygonal mesh. Curved surfaces cannot be expressed accurately using meshes, and can only be closely approximated using a high vertex density.

Instead of explicitly listing a number of vertices, there are ways to define surfaces purely mathematically. While GPUs are optimized to operate on meshes, a mathematical definition has certain advantages over the explicit approach using a mesh. Large meshes can take up a lot of memory by storing many vertices, edges, and faces, whereas mathematical definitions are often more compact. They also have practically infinite resolution, and can easily represent unbounded surfaces stretching to infinity.

Mathematically defined surfaces can be grouped into different types. Implicit surfaces are one type of mathematically defined surfaces, besides parametric surfaces.

3.1.2 Mathematical background

Implicit surfaces are all defined by a particular mathematical form [BB97]:

$$F(x, y, z) = c \tag{3.1}$$

The coordinates x, y, and z are used as functional arguments for the function F, which is called the "field function", assigning each point in the coordinate system a certain field value. F maps points from $\mathbb{R}^3$ to $\mathbb{R}^n$. $c \in \mathbb{R}^n$ is a constant, commonly set as 0. The implicit surface is defined by all points $X = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ where $F(X)$ evaluates to c. The different surfaces defined by varying the isovalue c are called the different "isosurfaces" of the field function.

This definition allows implicit surfaces to have a notion of an "inside" and an "outside". If for a point X it holds that $F(X) < c$, then the point is said to be on the inside, if $F(X) > c$ it is said to be on the outside, and if $F(X) = c$ the point is said to be on the surface.

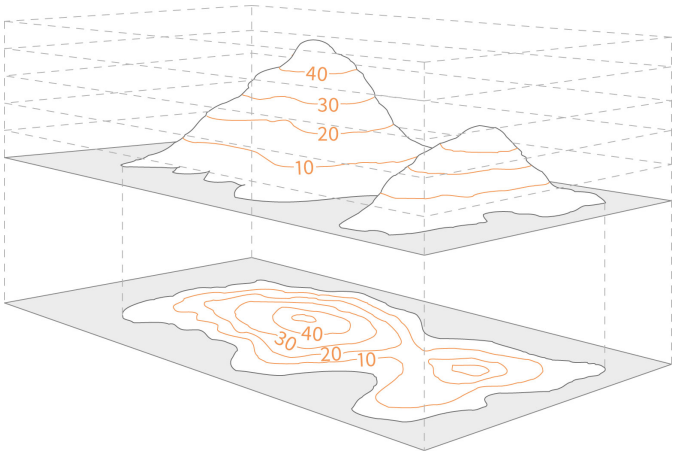


Figure 3.1: Contour lines as isolines. [Sur17]

A 2D analogue of field functions are elevation lines in topographic maps. Each point on a map has a certain elevation, and the elevation lines trace connected points that have the same elevation. In the 2D case they are called isolines instead of isosurfaces. The common elevation shared along the line is its isovalue. Figure 3.1 shows how topographic maps use isolines and how they relate to the 3D shape of a terrain.

The form of the field function F itself is not restricted by this definition and might be defined in a variety of ways. F could simply be a polynomial. But it is for example also possible to extract surfaces from CRT scan datasets, by having F sample from a data

set of uniformly spaced, discrete samples in a finite volume, interpolating field values between the samples.

3.1.3 Marching cubes

One basic idea to render implicit surfaces is to first extract a polygonal mesh that approximates the surface and then proceed to render the mesh with traditional methods. A popular algorithm to transform an implicit surface into a polygonal mesh is the marching cubes algorithm [LC87].

In this technique, a 3D volume is evenly subdivided into cubes. The field function that defines the surface to polygonize is evaluated only at the corners of the cubes. There are $2^8 = 256$ possible configurations for a cube and its corners to be in, as there are 8 corners and each corner has two possible states it can be in: It is either on or inside the surface, or entirely outside of the surface, using the definition of in- and outside introduced in Section 3.1.2. For each of these configurations a polygonal fragment is predefined that approximates the piece of the surface that intersects a cube with that configuration. If all corners have the same state (all inside, or all outside), then the surface does not intersect the cube, so no polygonal fragment needs to be inserted. However, if not all corners share the same state, then the surface must intersect the cube, and a polygonal fragment can be placed that separates the inside corners from the outside corners, approximating the surface. Figure 3.2 shows some possible cases for cube configurations, and the polygonal fragment that would be inserted. The corners marked with black circles are inside the surface, the others outside of it. The planes show the polygonal fragments that are inserted.

To save storage space, the number of cube configurations and their corresponding polygonal fragment that need to be stored can be reduced to a small number of base configurations by making use of rotational and reflective symmetry. The first published version of the marching cubes algorithm used just 15 base configurations. In some cases however, due to ambiguities these base configurations were not enough to identify a correct polygonal fragment. Later, a set of 33 base configurations were identified that could handle all cases without ambiguities [Che95].

For an improved reconstruction of the surface, the vertices of the chosen polygonal fragment can be moved along the cube edge, closer to where the actual surface should be, without having to perform additional field function evaluations. Since the vertex lies on an edge where both ends of the edge have different signs for their field value, the surface intersection can be approximated by linearly interpolating between the two corners to find where the field function would pass through zero. The corner whose field value is closer to the isovalue is likely to be closer to the surface, although that depends on the field function.

The polygonal fragments are fused together at the end to produce a mesh that approximates the surface. By choosing a higher sampling rate, the density of cubes increases and a better surface reconstruction is the result. However, that also increases the number

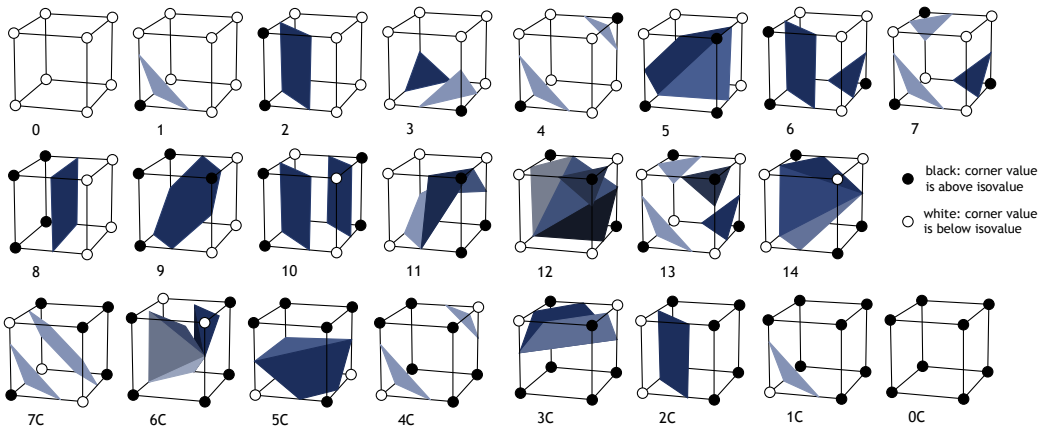


Figure 3.2: Example marching cubes cube configurations. [JC06]

of potentially expensive field function evaluations, as well as requiring more space to store the mesh. If the sampling rate is too low, the result is blocky and details can be missed entirely.

Improvements to the marching cubes algorithm have been made, for example by Wilhelms et al. by using an octree to subdivide the volume [WVG92]. Instead of subdividing the volume uniformly, the octree allowed them to subdivide the volume systematically, adding more detail where necessary, reducing detail in areas. Improvements like this lead to a faster algorithm and more space-saving meshes.

3.1.4 Raymarching

Raymarching is a technique to render implicit objects, such as implicit surfaces, directly [Tom12]. This can be done without having to first transform the surfaces into other representations such as polygonal meshes.

In this technique, the color of each pixel is determined by shooting a ray from the camera through a point in space corresponding to the pixels location on the screen, and checking at several points along the ray whether the object to render has been intersected. Once it is found that an intersection occurred, a more accurate intersection point can be found by iterative refinement, searching for the intersection point between the last two points checked until the desired accuracy is achieved. Once the intersection point is found, the distance of the object to the camera, and thus depth of the fragment, is known. Using the depth of the fragment, and the normal of the point on the surface, which can be calculated as the gradient of the field function, the fragment can be shaded and the pixel drawn.

To find the intersection point with an implicit surface, the field function is evaluated many times, starting at a point near the camera and continually at points marching

down the ray, until a point is evaluated as being on the inside of the surface, as defined in Section 3.1.2.

A step size parameter defines the spacing between the points along the ray at which the intersection test is performed. When the step size is small, the computational load is increased, as many more evaluations of the field function are required until an intersection is found. But if the step size is too big, then small sections of the object may be missed as the algorithm might step over it.

For other special forms of field functions, the step size can be chosen more efficiently. For example, for signed distance fields, where the value of the field function gives an estimate to the distance to the closest point of the surface, this distance can be used as a step size. That form of the algorithm is called "sphere tracing" and was first developed by Hart et al. [HSK89].

Raymarching is similar in concept to ray tracing, but can be used when intersections with the object are hard to solve for analytically.

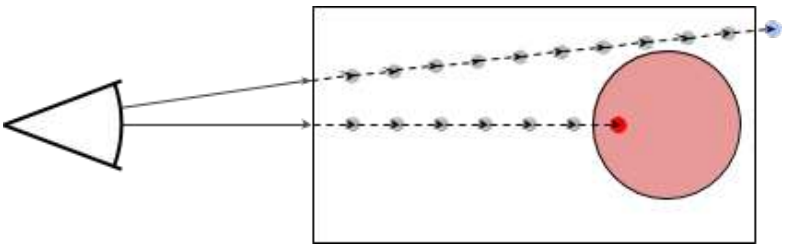


Figure 3.3: Raymarching visualized. [Zuc16]

3.2 Metaballs

Metaballs are a special type of implicit surface defined by particles. A powerful aspect of metaballs is that they can easily be combined, creating smooth blends between individual metaballs, as seen in Figure 3.4. As two metaballs get close to each other, they deform and merge into one bigger blobby object.

The field function that, together with a threshold value, defines the isosurface is the sum of the contributions. The contribution of a metaball particle is highest at its center location, and falling off towards zero with increasing distance. This falloff is typically not linear, but described by a Gaussian function. In Blinn's original definition, the contribution of a metaball particle was a Gaussian equation shown in 3.2.

$$f(x, y, z) = b \exp(-ar^2) \tag{3.2}$$

The function describes the particle's contribution strength at the coordinates x, y and z. r is the distance to the particle location, b is a scaling factor and a the standard deviation

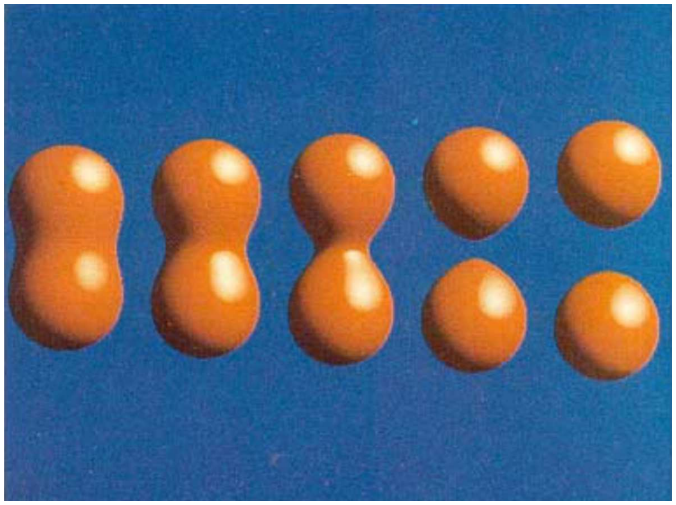


Figure 3.4: Metaball bonds stretching and breaking. [Bli82]

of the Gaussian. A single metaball particle in isolation would produce a spherical surface. By adjusting the parameters a and b one can adjust the radius of that sphere and the "blobbiness" of the resulting surface. The total field strength is simply the sum of the contributions by all metaballs, subtracting the threshold value T:

$$F(x, y, z) = \sum_i f_i(x, y, z) - T \tag{3.3}$$

Given the field function, the surface is defined by

$$F(x, y, z) = 0 \tag{3.4}$$

The original equation for the density field of a metaball (Equation 3.2) has infinite support, meaning it approaches but never reaches 0, and therefore has an infinite range of influence. Other equations are commonly used that have finite support, meaning they have only a limited range of influence. A finite support enables a number of optimizations by reducing the number of potentially expensive metaball density function evaluations necessary if it can quickly and efficiently be determined that a point in space is not influenced by a particular metaball. An example of a field function for an individual metaball with finite support is given by Equation 3.5. A graph for that equation can be seen in Figure 3.5.

$$f(x, y, z) = \begin{cases} (1 - r^2)^2 & |r| \leq 1 \\ 0 & otherwise \end{cases} \tag{3.5}$$

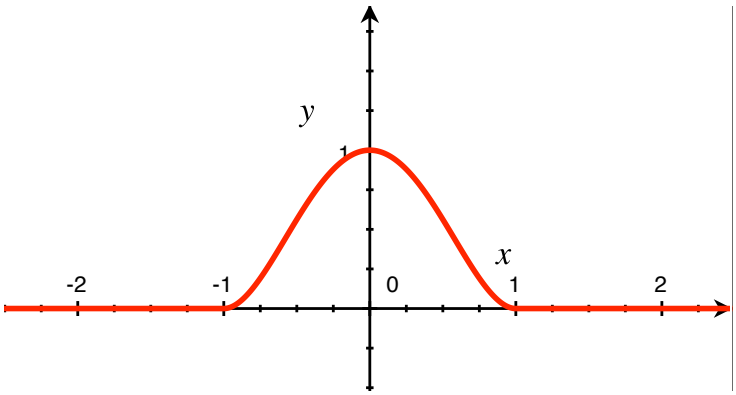


Figure 3.5: Graph of function described in Equation 3.5.

3.3 Artificial Neural Networks

Artificial neural networks are models of computation inspired by the human brain. A neural network is made up by a number of artificial neurons, connected together in specific ways. One big aspect of neural networks is their ability to "learn". This section will explain the building blocks of neural networks.

3.3.1 Perceptron

Psychologist Frank Rosenblatt sought to model a biological neuron mathematically. He named the model he created "Perceptron" [Ros58].

A biological neuron receives signals from neighboring neurons via its dendrites, and outputs a signal to other neurons via its axon if the neuron reaches an activation potential and fires. In Rosenblatt's model, a perceptron is a computational unit that takes multiple inputs that can be either 0 or 1. The output of the perceptron is a weighted sum of the inputs with an "activation function" applied to it. The weights model the strength of the neuronal connection between the perceptron and its input. These inputs can act as either excitatory if the value is positive, or inhibitory if the value is negative. A bias value is used as a means to add an offset to the weighted sum.

Formally, there are i inputs to the perceptron labeled x_i, with associated weights labeled w_i. The bias value b is added to the weighted sum of the inputs, and the activation function f is applied to the sum. The equation describing the output of a perceptron can be seen in Equation 3.6.

$$f\left(\sum_i w_i x_i + b\right) \tag{3.6}$$

Rosenblatt's perceptron used a step function as activation function. When the input of a step function crosses a threshold value, then its output is 1, otherwise it is 0,

meaning it is all-or-nothing. This activation function models the action potential required for a biological neuron to fire. Only when the weighted sum is large enough does the neuron trigger and send a signal itself. Later artificial neurons also used other activation functions, the sigmoid function, having an s-shaped curve, being a popular choice. Other activation functions are described later.

A perceptron acts as a binary classifier. Depending on the pattern of its inputs, it may classify the inputs as belonging to either Category 0 or Category 1. By adapting the weights and biases, the hope was that the perceptron could classify a wide range of different patterns. However, Marvin Minsky and Seymour Papert showed in their book "Perceptrons: An Introduction to Computational Geometry" [MP69] that perceptrons are only able to classify linearly separable patterns. They demonstrated that a perceptron is able to reproduce the logic functions OR and AND, but unable to reproduce the XOR function.

3.3.2 Multilayer Perceptrons

Minsky and Papert also showed that by chaining together multiple layers of perceptrons, nonlinear functions such as the XOR function could be represented [MP69].

This multilayer architecture has the ability to serve as universal approximator - it can approximate any continuous function, as shown by Hornik et al. in 1991 [Hor91].

The general structure of multilayer perceptrons is shown in Figure 3.6. They consist of an input layer, an output layer, and a number of hidden layers in-between. Each layer's output is the input of the next layer. In fully connected layers, each perceptron is connected to each perceptron of the next layer.

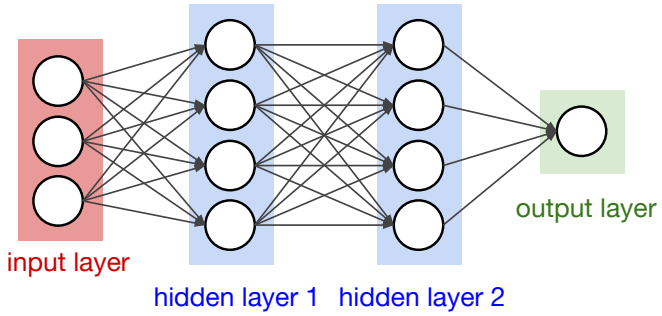


Figure 3.6: Multilayer perceptron. [CS219]

Even though multilayer perceptrons are universal approximators, an issue was that they were difficult to train. With simple perceptrons, changing the weights and biases directly influenced its output. But changing the weights and biases in multilayer perceptrons only has an indirect effect on the output. This, together with the increased number of parameters, meant that adjusting the parameters to match the desired behaviour was a lot more difficult.

3.3.3 Backpropagation

The backpropagation algorithm enabled neural networks to be trained effectively. The algorithm was derived in the 60s by multiple researchers [Sch15], although its application to training neural networks was not realized until popularized by Rumelhart et al. in 1986 [RHW86]. Linnainmaa first implemented the backpropagation algorithm on a network of nodes in his thesis in 1970 [Lin70], although he did not yet explicitly mention neural networks.

The name "backpropagation" stands for backwards propagation of error. In supervised machine learning, the neural network is presented with labeled training examples for which the expected output is known. The "error" is the difference between the neural network's current output (the outputs of the neurons in the output layer) and the expected output. This error is also called the loss value of the network, and it depends on the weights and biases of the network. In the algorithm, the gradient of the loss function, with respect to the weights and biases, is calculated. The gradient of a function corresponds to the direction of steepest incline, from which you can derive how to update the weights and biases in order to decrease the loss value. By propagating the error backwards and calculating gradients along the way, the weights and biases of the entire network can be updated to reduce the overall error.

One iteration of gradient descent takes all training examples into consideration, calculating the best way to update weights and biases to reduce the overall loss over all examples. However, with the huge data sets now common in machine learning, this can take a very long time. Stochastic gradient descent chooses random training examples and calculates the gradient for these single examples. This gradient only approximates the total gradient, but given enough training examples the approximation is good, even if the process is noisy.

A compromise between full batch gradient descent and stochastic gradient descent is offered by mini-batch stochastic gradient descent. This variant of the algorithm takes small batches, commonly between 50 and 256 [Rud16], of randomly chosen training examples to calculate the loss function gradient. This smoothes out the gradient compared to using single training examples, reducing the noise in the weight updates, while still being much faster to calculate than a full batch gradient descent.

3.3.4 CNNs

Fully connected multi-layer neural networks as discussed so far can be used to classify images, such as images of handwritten characters or animals. But images are typically large. A 10×10 pixel black and white image would already contain 100 units in its input layer, each unit corresponding to a pixel in the input image, and several tens of thousands of weights [LHBB99]. With such a large number of parameters, training can take a long time and requires a large training set. A bigger issue than that however, for the task of classifying images, is that these networks do not handle translation or scaling of the input well. Furthermore, fully connected layers ignore the topology of the input,

so do not make use of the fact that pixels in 2D images that are close to another are highly correlated. Convolutional neural networks are designed to fix these issues by using a receptive field of units that are close together.

Convolutional layers

Convolutional neural networks, or CNNs for short, consist of several convolutional layers. These differ from fully connected layers in the way its units are connected to units from the previous layer. Only the corresponding unit from the previous layer and its surrounding units are used as input for that unit. The basic building block of such layers is the convolution operation. Image convolutions have long been used in computer vision for a variety of purposes, such as edge detection [Sob14] and blurring, among others.

The convolution operation takes a source image and a convolution kernel and produces an output image. Each pixel in the output image is computed as the weighted sum of the corresponding pixel and its surrounding pixels in the source image, where the weights are defined by the convolution kernel, as seen in Figure 3.7.

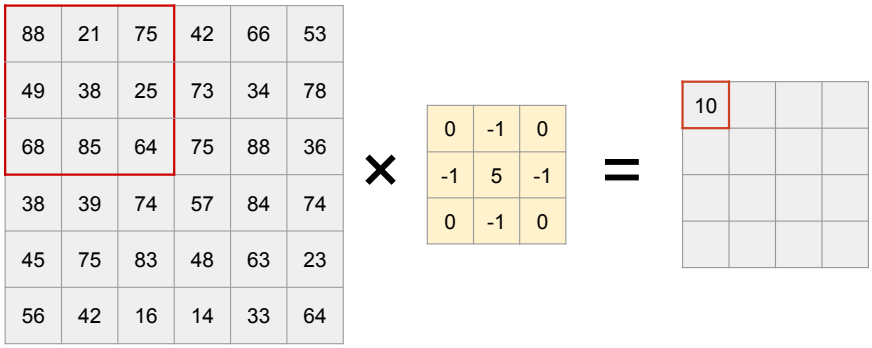


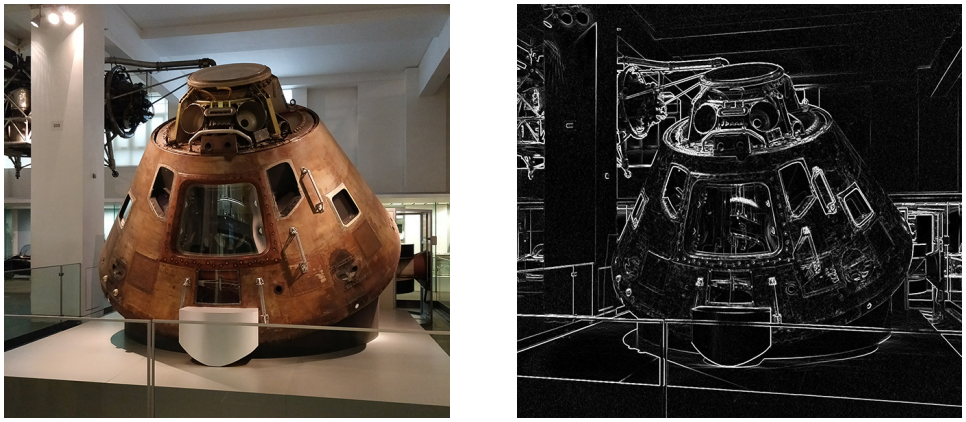
Figure 3.7: Convolution operation.

An example of a convolution filter used in image processing is the Sobel edge-detection filter [Sob14]. The kernel of the Sobel filter for detecting vertical edges is shown in Figure 3.8. In Figure 3.9 you can see an example of the application of the Sobel edge-detection filter. In the output image edges are highlighted by the filter.

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

Figure 3.8: Sobel filter kernel G_x.

Convolutional layers in neural networks are wired in a way that reproduces the image convolution operation. Each unit in the convolutional layer only receives inputs from the corresponding unit and its surrounding units in the previous layer. This group of units is



(a) Original image (b) Convolved image

Figure 3.9: Applying the Sobel edge detection kernel to an image.

called the unit's "receptive field". The weights of the connections, which correspond to the convolution kernel weights, are shared across all units, ie. the same convolution kernel is applied across the entire input. This reduces the number of parameters greatly compared to a fully connected layer, only requiring a number of parameters that corresponds to the size of the convolution kernel instead of weights for connections between all units. Convolutional layers are also robust against translation. If a feature in the input image is translated, the resulting feature map is also just translated.

The input to convolutional layers can also be thought of as a 3D volume instead of a 2D image, with each pixel having multiple components. The filter kernel matches the dimensionality of the input volume. Instead of learning just one convolution filter kernel, a convolutional layer can also be used to learn several kernels, producing an output volume. The images produced by a single filter kernel is also called a feature map. Thus a convolutional layer could learn several filters producing feature maps that highlight different features such as edges and corners.

Additional meta-parameters (parameters that are not learned but fixed in the design of the network) for a convolutional layer are kernel size, stride, padding. Stride controls how the filter convolves around the input volume. A stride of one shifts one unit at a time, causing overlapping receptive fields. "Padding" controls what happens when convolving at the borders of the input. Since there are no neighbors outside the border of an image a replacement value has to be used. This can for example be computed by using a fixed value like 0, by using the nearest available value from the image, or by mirroring the input.

Sub-sampling layers

Convolutional layers are usually followed by sub-sampling layers that reduce the size of the input. The idea is to drop unnecessary fine detail and reduce the number of neurons,

and thus fewer parameters that are required.

A common type of sub-sampling layer is the max-pooling layer, depicted in Figure 3.10. For a given square of neurons in the source layer, only one neuron exists in the sub-sampling layer. The neuron outputs the maximum value of all the input neurons in the corresponding square. For a 2×2 pooling layer the size of the output would only be half that of the input, but still keep all the greatest activations in the layer.

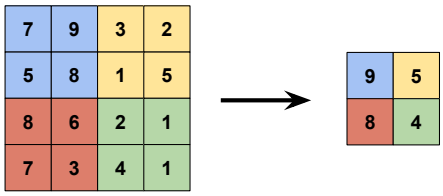
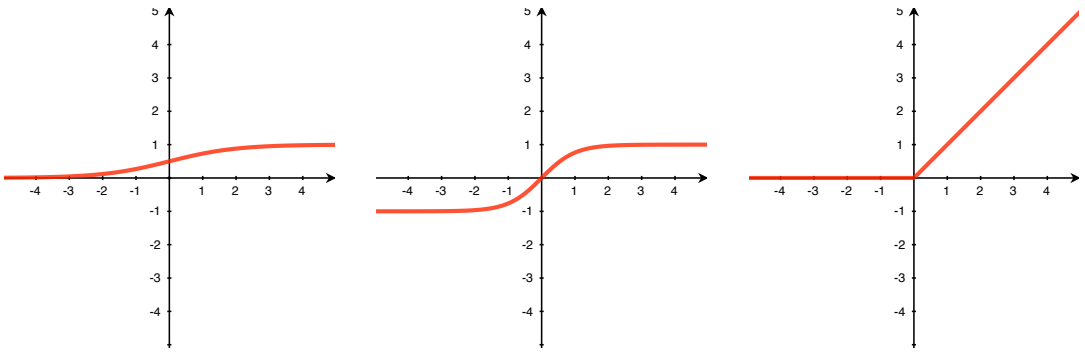


Figure 3.10: Max pooling operation.

Non-linear activation functions



(a) Sigmoid (b) tanh (c) ReLU

Figure 3.11: Graphs of different activation functions.

In Section 3.3.1 the activation function for perceptrons were described. Activation functions introduce non-linearity into a process that so far computed only a linear combination of values, and are often appended after convolutional or subsampling layers.

The sigmoid non-linearity has the mathematical form $\sigma(x) = 1/(1+\mathrm{e}^{-x})$ and is shown in Figure 3.11a. It takes a real valued input and produces an output in the range between 0 and 1. Large negative numbers become 0 and large positive numbers become 1. It was used frequently early on, as it behaves in a way that is reminiscent of biological neurons: from not firing at all to fully-saturated firing at full signal strength.

The tanh non-linearity is similar to sigmoid function. A graph of the function is shown in Figure 3.11b. It has an output range of $[-1, 1]$, and unlike the sigmoid function, the output of the tanh function is zero-centered.

Both the sigmoid and tanh activation function can suffer from a problem called "vanishing gradients". Since the weight update in neural networks using back-propagation depends on gradient descent, if the gradient of a function is small then the update to its weights will also be small. If sigmoid or tanh neurons get saturated at either of its extremes, then learning can become very slow or entirely stop.

A ReLU layer (Rectified Linear Unit) [XWCL15] is often appended after a sub-sampling layer. Its graph can be seen in Figure 3.11c. The ReLU layer applies a function to its input that clamps all its negative values to 0, leaving positive activations unchanged. It is represented by the function $f(x) = \max(0, x)$.

However, ReLUs can also get into an inactive state by too large weight updates. "Dying ReLUs" are not be able to update their weights anymore and continue to give the same output over all future iterations. The dying ReLU problem happens when the gradients are too large, the vanishing gradient caused by the Sigmoid function actually helps in preventing ReLU units from dying during back-propagation. This motivates the use of Sigmoid functions at the two ends of the network, and the ReLU for other layers [CSAT17].

Leaky ReLUs [XWCL15] are one attempt to fix the "dying ReLU" problem. Instead of the function evaluating to zero when the input is negative, a leaky ReLU will have a small negative slope instead. It can be expressed through the equation $f(x) = max(\alpha x, x)$ where α is a small constant, so that a small gradient exists in the negative range.

Image classification with Convolutional Neural Networks

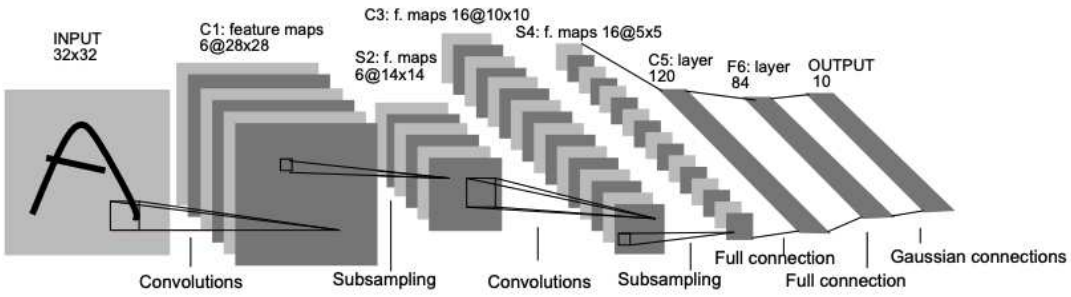


Figure 3.12: LeNet architecture. [LHBB99]

We looked at some building blocks of convolutional neural networks: convolutional layers, sub-sampling layers and the ReLU layer. With these building blocks discussed, we can now examine the network structure of a full convolutional neural network. As an example, we will look at Yann LeCun's LeNet5 [LHBB99] network for character recognition.

The diagram of the architecture can be seen in Figure 3.12. It consists of multiple convolutional layers, followed by sub-sampling layers. As the input is reduced in size, the

volume increases in depth, calculating more and more higher-level feature maps based on the lower level features detected in the previous layers. The first layers typically learn simple features like edges, later layers learn abstract higher level features based on these lower level features. CNNs used for classifying animal pictures might build higher level features corresponding to pointed ears or tails. The last layers of the network are fully connected layers used for classification. The output layer consists of as many units as there are classes to recognize and is trained to be one-hot encoded (only the unit corresponding to the actual class should have a value of 1, all other units should have a value of 0).

Convolutional neural networks have first been used for optical character recognition, but have become successful for a wide variety of image classification tasks, as well as speech recognition, translation, part-of-speech tagging, and more.

3.3.5 Generative Adversarial Networks

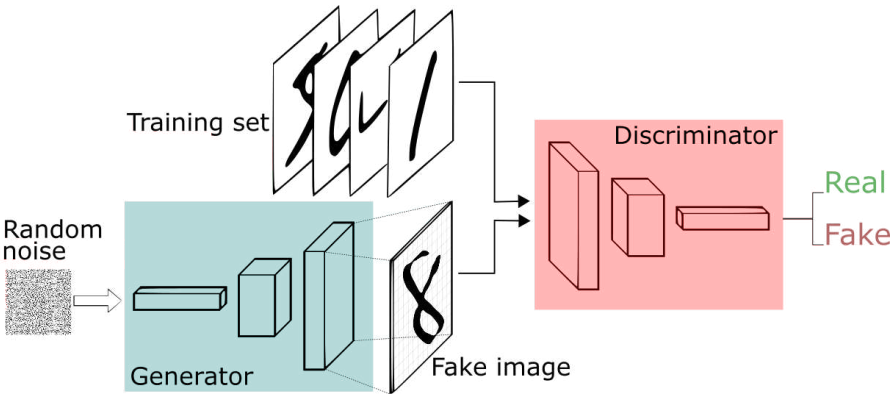


Figure 3.13: Architecture of GANs. [Sil18]

Generative Adversarial Networks, or GANs for short, are a type of architecture for neural network that has led to significant improvements in image generation, video prediction, as well as tasks from a number of other domains [ZML16]. They were first described by Ian Goodfellow in 2014 [GPAM+14].

GANs consist of two parts - a generator network and a discriminator network that are trained concurrently. The discriminator network is a classifier network, using a convolutional neural network to learn to classify an image as either "real" or "fake", as described in Section 3.3.4. In this case, "real" means the image comes from the data distribution it was trained on, and "fake" that the image does not come from that data distribution. The output of the discriminator is a probability that the input came from the real data distribution.

The goal of the generator network is to produce images that fit the data distribution that the discriminator was trained on, and to fool the discriminator into thinking it is real. Generally, to produce these distributions, a deconvolutional neural network is used as

generator [CJC+19]. Deconvolutional networks consist of deconvolutional layers. The transposed convolution operation, called deconvolution in most literature, can be seen as the backward pass of a corresponding traditional convolution. Contrary to the traditional convolution that connects multiple inputs to a single output, deconvolution associates a single input with multiple outputs [GWK+18]. The outputs of deconvolution layers are larger in size than its inputs, having an upsampling effect. The generator is fed with a randomized input vector, often a 100-dimensional vector with Gaussian distribution [CJC+19].

During the training of GAN, the parameters of the generator and the discriminator are updated iteratively. When the generator is being trained, the parameters of the discriminator are fixed, and when the discriminator is trained, the parameters of the generator are fixed [CJC+19]. The data generated by the generator is labeled as fake and is input into the discriminator. The generator receives the gradient of the output of the discriminator with respect to the fake sample and uses it to update its own parameters [ZML16]. In that way the discriminator guides the generator to produce better images.

It is notable that the generator can be any algorithm as long as it can learn distribution of training data, and the discriminator needs to extract features and train a binary classifiers using these features. For example, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and long-short-term memory networks (LSTMs)[CJC+19].

The two networks are pitted against each other, each trying to outperform the other. That is why they are given the name "adverserial" network. As each network improves, it drives the opposing network to improve its methods, until the fake samples are indistinguishable from real samples. The GAN framework is inspired by minimax two-player game. When the generated data is input to the discriminator, the goal of the discriminator is for its output to be 0 (representing zero probability that the generated sample comes from the training distribution), while the goal of the generator is to make the output of the discriminator 1. When two models have been sufficiently trained, the game eventually reaches a Nash equilibrium. Ideally, the discriminator cannot tell whether the input is real data or generated data [CJC+19].

It can be shown that the generator learns representations instead of just memorizing picture elements from a database [CJC+19]. Radford et al. 2015 [RMC15] introduced deep convolutional generative adversarial network (DCGAN). They demonstrated their network was able to learn representations of faces and assign meaning to its input vector components. That allowed them to demonstrate interesting vector arithmetic on faces, revealing rich linear structure in representation space. For example, by taking the average of input vectors that produced men with glasses, subtracting the average of input vectors that produced men without glasses, and adding the average of input vectors that produced women without glasses, the resulting input vector generated an image of a woman with glasses, as seen in Figure 3.14. They also demonstrated the ability to interpolate between faces by interpolating between their input vectors.

GANs are however hard to train and can exhibit some problems, such as mode collapse and

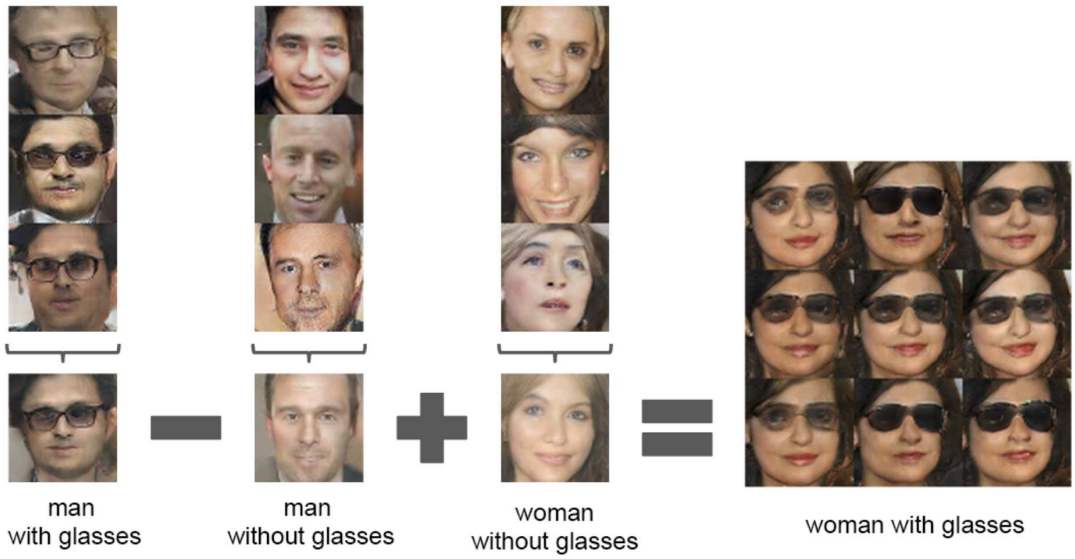


Figure 3.14: Vector arithmetic on faces. For each column, the input vectors of samples are averaged. Arithmetic was then performed on the mean vectors creating a new vector Y. The center sample on the right hand side is produced by feeding Y as input to the generator. To demonstrate the interpolation capabilities of the generator, uniform noise sampled with scale +-0.25 was added to Y to produce the 8 other samples. [RMC15]

non-convergence [CJC+19]. Real-world distributions are highly complex and multimodal. Mode collapse is a failure mode of GANs in which the generator generates a limited diversity in samples, maybe even the same sample, regardless of its inputs.

After their introduction, GANs have garnered a lot of attention. Several network architectures have been proposed based on GANs, such as deep convolutional neural networks (DCGAN) [RMC15], conditional GANs (cGAN) [MO14], Wasserstein GAN (WGAN) [ACB17], Energy-Based GANs (EBGAN) [ZML16] and more. Besides generation of high quality images learned from a data distribution, typical applications for GANs in computer vision also include style transfer and image translation, image inpainting and super resolution.

3.3.6 Conditional GANs

An extension to GANs are the conditional generative adversarial networks (cGANs) proposed by Mirza et al. in 2014 [MO14]. In unconditioned GANs with large or complex data distributions, it is difficult to control the generated results. To guide the generation process in a cGAN, additional information is supplied to the generator and discriminator. The structure of cGANs is shown in Figure 3.15.

The additional information is called the condition. Anything may be used as condition, such as a class label, some part of the desired output data or text. The discriminator

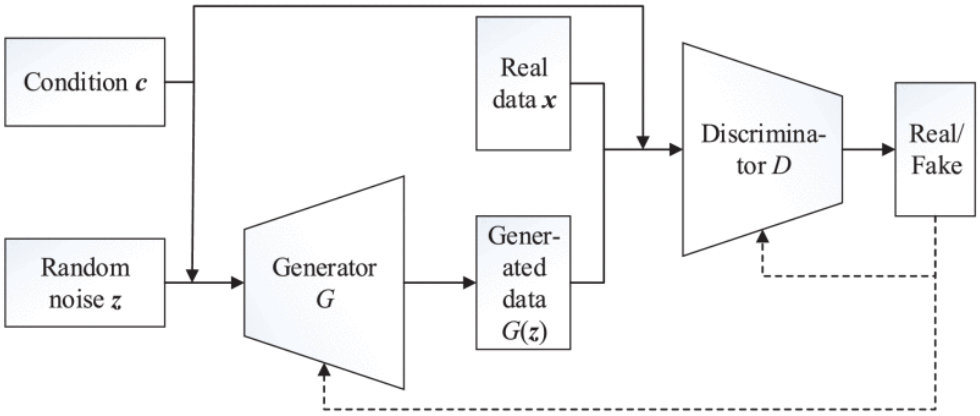


Figure 3.15: Architecture of cGANs. [CJC+19]

decides whether a sample looks "real" given the condition, and the generator tries to generate a sample from the data distribution, given the condition.

This type of network has been used for image-to-image or style transfer tasks. In that case, the image that is to be translated to another image is used as the condition. Examples of style-transfers are shown in Figure 3.16.

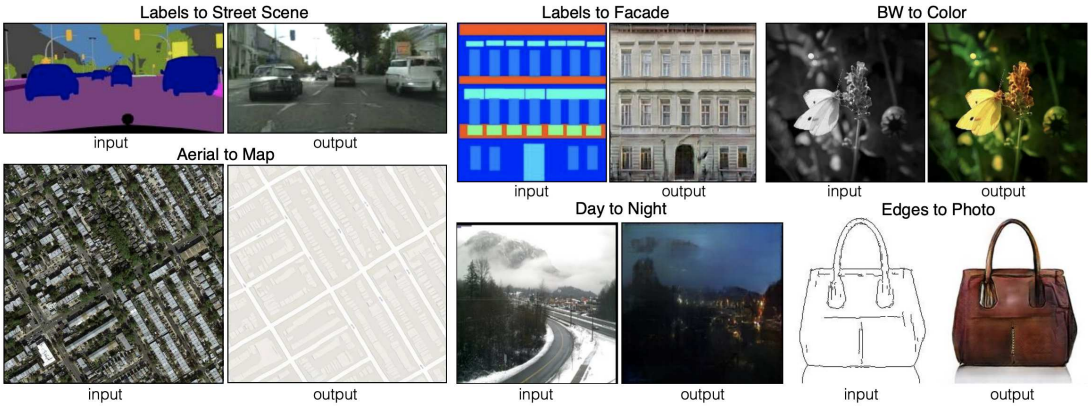


Figure 3.16: Examples of image-to-image translation. [IZZE17]

3.4 Deferred shading

When rendering a 3D scene as an image, one needs to take proper care of handling obstruction of objects. Surfaces that are hidden by other objects should not be visible in the final image. This problem is called hidden surface removal. One of the earliest algorithms to deal with this is known as the painter's algorithm, or priority fill. It works much in the same way a painter paints a scene on a canvas, by drawing distant objects first, followed by closer ones. To do this, the polygons that need to be drawn are first ordered by their depth (distance to the camera). As the polygons are drawn back to front, the fragments of the polygon are shaded, taking into account material properties, textures, and lighting information. The result is written to the frame buffer, overwriting pixels that have previously been written [Nyb11]. While being simple, the painter's algorithm has several weaknesses. For example, it requires special treatment of polygons with cyclic overlap (A occludes B, B occludes C, C occludes A) or polygons piercing one another.

One algorithm that solves these issues is called depth-buffering, or z-buffering. Depth buffering works by introducing another buffer which keeps track of the depth of all pixels that are drawn. Before shading calculations for a fragment are performed, the depth buffer is first checked for occlusion. This is called the depth test. Only if the fragment is known to be closer to the camera than the previously drawn pixel at that location the pixel is drawn, otherwise the fragment is discarded. When a pixel is drawn, the depth buffer is updated with the new depth information. Another advantage of depth buffering over the painter's algorithm is that it does not require the polygons to be sorted before drawing. In fact, polygons can be drawn as soon as they are available, without having to wait for all polygons from a scene to be available. Transparent objects need special handling though, as a fragment that is behind a transparent fragment still needs to be visible. Usually transparent polygons are marked and rendered separately in a second rendering pass.

A problem that causes wasted computation is called overdraw. Overdraw refers to the same pixel being drawn multiple times. The computation that went into shading a pixel is wasted when it is overwritten. The painter's algorithm usually produces a lot of overdraw. Depth buffering usually reduces overdraw, but is not exempt from it. It can have considerable cost if there is a lot of overdraw and shading calculations are complex.

Deferred shading is a technique that reduces wasted computation due to overdraw. Instead of computing the shade of a fragment when it is drawn, shading computation is deferred as it might be discarded later if occluded by another fragment. To be able to perform shading later, the attributes required for the computation, such as world coordinates, material properties such as roughness and diffuse color, and normals are stored in separate off-screen buffers. The set of attributes that need to be stored depends on how the shade should be computed, and can be different for different shading effects. The buffers used to store the attributes are commonly called G-buffers, short for "geometric buffers" ([AMHH08], [ST90]). Figure 3.17 shows examples of such buffers.

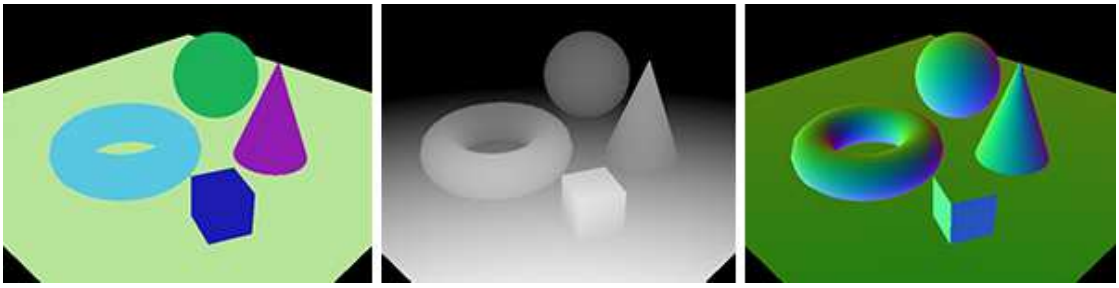


Figure 3.17: G-Buffers: Diffuse color, depth, and normal buffers. [Wik19]

In the first rendering pass, the attributes are collected and the G-buffers are filled. The final image is composited in a second rendering pass, by issuing a full-screen quad to be drawn that combines the G-buffers accordingly.

Deferred shading can be combined with other techniques by using the depth buffer to compose scenes. By keeping the depth buffer from the first scene, another scene can be rendered into the image, using depth buffering to handle occlusions correctly.

CHAPTER 4

Conceptual Framework

In this chapter, we present the planned approach to generate images of approximate metaballs with the help of neural networks. The network design is based on the work of Isola et. al, "Image-to-image translation with conditional adversarial networks" [IZZE17]. A neural network needs a dataset to be trained on, so the first section will discuss the generation of our dataset. In the following section, we will discuss the network design itself. Lastly, we discuss how to use the network in a larger context.

4.1 Generating a dataset

In this section we discuss the dataset we need to train the neural network. The network used for this thesis is based on an image-to-image translation network, so the input itself will be an image. No public data set for this task exists, so we generated one. The requirements for the input as well as target output images, and how they are generated, are described in the following.

To reduce the number of variables to take into account and simplify the model, some restrictions have been placed on the data set. All images generated use the same field function and threshold values, and the parameters for the perspective projection used to render the images is fixed.

4.1.1 Inputs

The input image is used as a source for the image-to-image translation. It should represent the metaballs to be drawn, and will be the condition in the cGAN that guides image generation (see Section 3.3.6).

One key feature of the input images has to be that it contains all the information needed to produce an acceptable output. The second key feature is that, in a real-time application, the input image can be constructed quickly and efficiently.

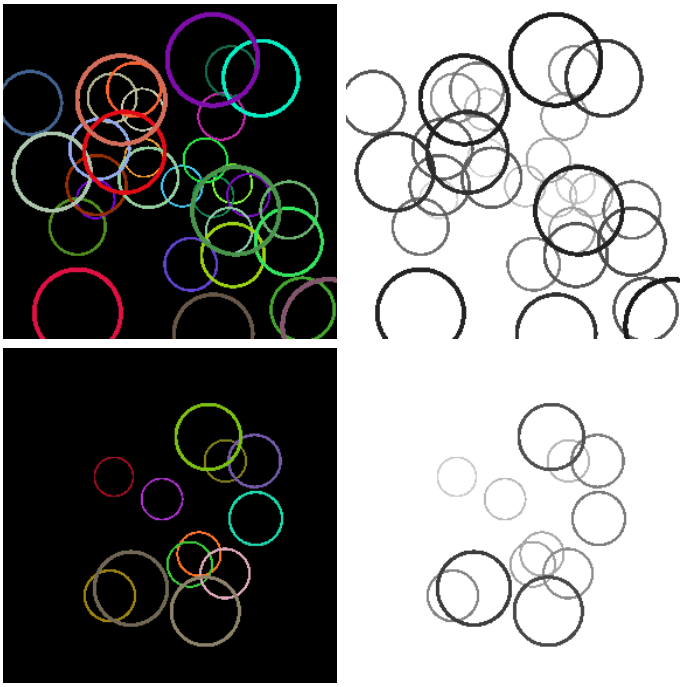


Figure 4.1: Example input images, with circles representing metaballs. The depth channel is represented separately in the right column, with black being close to the camera and white being far away.

In this work, the representation chosen for the input images consists of circles placed with their centers at the location of the metaball particles. The radius of the circles is chosen to be close to the minimal radius that the corresponding metaball in isolation would have when drawn with perspective projection. This way, the input image has similar structure to the output image, in that the location and approximate size matches between them. Image-to-image translation neural networks can make use of the spatial structure of the input image, as will be described in more detail in Section 4.2.1. If solid discs were drawn for each metaball, instead of circles, then metaballs farther away could become completely occluded. But these metaballs, even if they would be hidden if each metaball was drawn as its minimal sphere, can still contribute to the field value in its surrounding area. Since it can have an effect on the shape of non-occluded metaballs it must also be represented in the input image. By only drawing outlines of circles for each metaball, metaballs don't become occluded, and their information is not lost.

The input image has four channels: three color channels (rgb) and a depth channel. The color a circle is drawn in represents the diffuse base color of the metaball, the depth corresponds to the depth of the center of the metaball. Examples of such inputs can be seen in Figure 4.1.

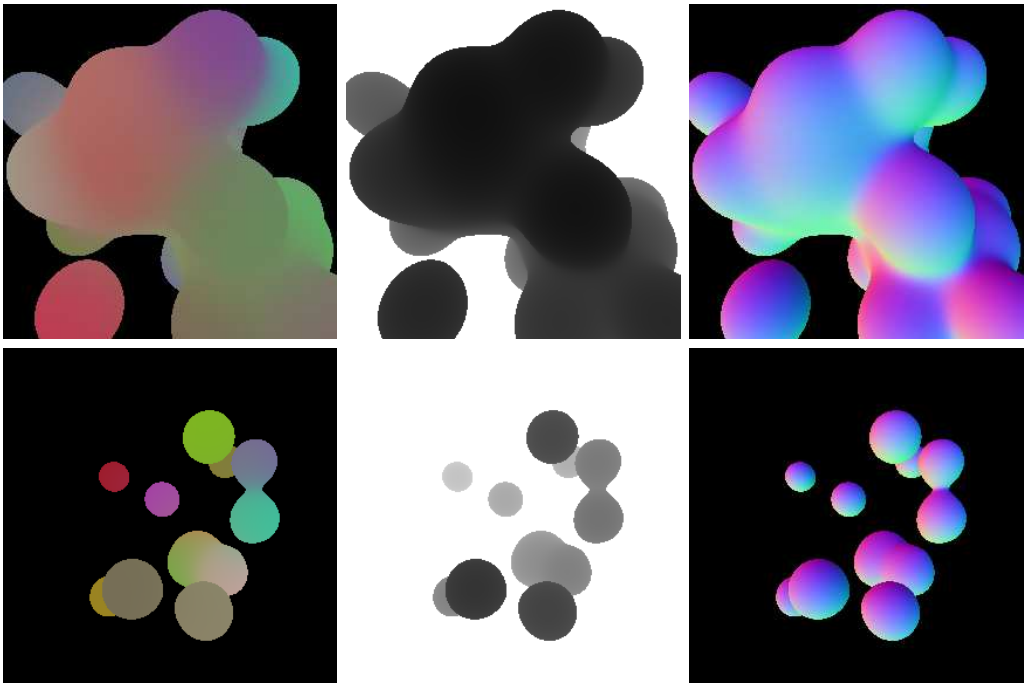


Figure 4.2: Example output images, with color buffer, depth buffer and normal buffer depicted separately.

4.1.2 Outputs

The output, or target image, will be an image of the blended metaball shapes. To make the process independent of lighting, we chose to output not a final composited image, but instead G-buffers as used in deferred shading (see Section 3.4). In particular, we want to output a depth buffer, a normal buffer, and a diffuse color buffer.

The depth buffer will be a 1-channel image containing the depth values, with near being chosen as 0 (black) and far chosen as 1 (white), using a linear scale throughout the depth range.

The normal buffer will be a 3-channel image containing the encoded normal of the surface, or 0 if there is no surface at that location. The three components of the normal buffer correspond to the x, y, and z components of the normalized normal vector, but projected from the range -1 to 1 to the range 0 to 1.

The diffuse color buffer will be a 3-channel image containing the color of the metaballs, where the color is blended across the metaballs, based on the proportion of contribution to the field function from nearby metaballs. The channels will represent the red, green and blue components of the color.

These buffers can be combined to a final image in a compositing step at a later stage, including lighting calculation, using the deferred shading technique described in Section

3.4. To render the target output images, an exact method like ray-marching or fine marching cubes can be used. Examples of target output images can be seen in Figure 4.2.

The samples in the training set we generated contain a randomized number of metaballs, randomly placed in a volume in front of the camera. For each sample, both input and output images are generated together as a pair.

4.2 Network architecture

We base this work on the network architecture presented by Isola et al. in "Image-to-Image Translation with Conditional Adversarial Nets" [IZZE17]. This network is a type of conditional generative adversarial network (see Section 3.3.6), consisting of two major parts, a generator and a discriminator.

4.2.1 Generator

The generator part of the network can be split into two halves. The first layer receives the input image as conditional input (see Section 3.3.5), represented as an image with 4 channels (see Section 4.1.1). Convolutional layers extract features from the conditional input. The second half of the generator is a deconvolutional neural network used to produce an image from the features and conditional input.

In image-to-image translation, high resolution input images are mapped to high-resolution output images. While the output images differ in surface appearance, both input and output images share a lot of their underlying structure. The generator architecture makes use the assumption that the structure of the input image resembles that of the output image by being designed as a "U-Net" [RFB15] with added skip connections [IZZE17]. To share information between layers from the convolutional part and corresponding layers from the deconvolutional part of the generator network, skip connections allow the deconvolutional layer to inspect the output of the convolutional layer directly. Specifically, skip connections between each layer i and layer $n - i$ are added, where n is the total number of layers. Each skip connection simply concatenates all channels at layer i with those at layer $n - i$ [IZZE17]. This structure is depicted in Figure 4.3.

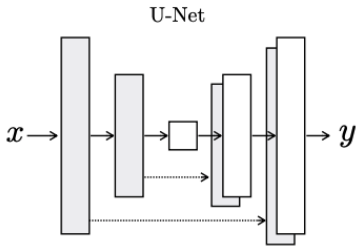


Figure 4.3: U-Net Generator Architecture. [IZZE17]

4.2.2 Discriminator

The second part of the overall network architecture is the discriminator. Following the structure of a conditional GAN the discriminator also sees the input image (see Section 4.1.1) and decides whether the output image matches the input it is conditioned on. The discriminator learns to distinguish "real" images from "fake" images, where "real" in this case means generated by the exact rendering method, and "fake" meaning being generated by the generator part of the network.

4.3 Embedding the network into a renderer

Once a data set is generated and the network trained on the data set, the network is ready to be used. We implemented a realtime renderer that uses the network to draw metaballs. For each frame to render, the simple circle input image from raw metaball data is drawn and fed into the network. The output of the network consists of G-buffers that can then be combined using deferred shading to produce a final composited image to display on the screen. In Chapter 5.4 the implementation of our demo-renderer is described.

CHAPTER 5

Implementation

In this chapter we will describe the details of the implementation. First, an overview of the used software and frameworks is given. In the following section it is described how the training examples for the neural network were generated. After that, the network architecture is described in detail. Lastly, the implementation of the demo renderer is explained.

Source code for the entire implementation is available online[1].

5.1 Software and Frameworks

The system was implemented on a PC running the Windows 10 x64 Pro operating system by Microsoft. To utilize GPU acceleration for machine learning tasks on the Nvidia hardware that we used, we installed CUDA 9.0 and cuDNN (NVIDIA CUDA Deep Neural Network library) 7.0.5. CUDA is platform for parallel computing, and cuDNN is a GPU-accelerated library of primitives for deep neural networks. Amongst the variety of available machine learning frameworks, Tensorflow was chosen to model, train and execute the neural network. Tensorflow is a cross-platform open source machine learning system released by the Google Brain team in 2015 [ABC+16]. Models defined using Tensorflow can be run on many different types of machines and computational devices, including multicore CPUs and general purpose GPUs. The computation can also be spread out over a cluster of servers. Tensorflow on Windows uses CUDA and CuDNN for its GPU acceleration.

All code written for this thesis was written in Python 3.6.2. The python packages that have been used are listed in Table 5.1. GLFW is an OpenGL library, used together with the FreeGLUT library and pyopengl to perform drawing using OpenGL. Pillow is a fork

[1] https://www.gitlab.com/robhor/deep-metaballs

of the Python Image Library (PIL), used for some image manipulation tasks. OpenCV is a library for computer vision tasks that has also been used for image manipulation. Finally, NumPy is a library for scientific computing, and supports handling of large multi-dimensional matrices.

Package name	Version
tensorflow-gpu	1.10.0
tensorboard	1.10.0
glfw	1.7.0
pyopengl	3.1.1a1
Pillow	5.1.0
opencv-python	3.4.2.17
NumPy	1.14.5

Table 5.1: Python packages used.

5.2 Example generation

Before the neural network can be trained, data is required to train it on. This includes both inputs and the target outputs for the model. A synthetic dataset for our application can easily be generated, since we do not rely on real world data.

Python scripts were used to generate inputs and target outputs used for training. For each training example a random number of metaballs were generated, each with random position and color. OpenGL was used to render the images. The input images consist of quads for each metaball used to render their circle, and the output images were rendered using a ray-marching technique. The total number of examples used to train the network in our case was 20000. A relatively small example training set was deemed sufficient, as the results were in line with expectations, adding more examples saw diminishing returns.

In the next sections we will detail how both input and output images were generated, and then brought together into a format that the training script could use.

5.2.1 Example generator framework

The main driver for training example generation is a python script. A GLFW window is set up to render the example images in using OpenGL. In a loop, new example parameters are set up, and then handed to a renderer for output images, and input images. To capture the effects of small changes in the input to the resulting output images, each example of metaballs is rendered three times, with small random offsets are applied to

the metaballs each time. After an image is rendered, the image is read back from GPU memory using a `glReadPixels` call and written as a PNG file to disk using the Pillow image processing library. Code for this outer loop is shown in the following listing:

```
def main():
    window = glfw.create_window(size[0], size[1], "Generator", None, None)
    glfw.make_context_current(window)
    glfw.swap_interval(1)

    while not glfw.window_should_close(window):
        display()
        glfw.poll_events()

    glfw.terminate()

def display():
    for i in range(0, GENERATE_COUNT):
        balls = generate_balls()
        out_path = os.path.join(outDir, str(i).zfill(6))
        draw_balls(balls, out_path)
        glfw.poll_events()
    sys.exit(0)
```

For each iteration, the example scene is drawn using different renderers:

```
def draw_metaballs(balls, out_path):
    glfw.poll_events()
    os.makedirs(out_path, exist_ok=True)

    def swap_and_save(name):
        glfw.swap_buffers(window)
        save_buffer_as(out_path + "/" + name + ".png")

    objectMetaballsRenderer.render_color(balls)
    swap_and_save("obj_color")

    objectMetaballsRenderer.render_depth(balls)
    swap_and_save("obj_depth")

    objectMetaballsRenderer.render_normals(balls)
    swap_and_save("obj_norm")

    objectMetaballsRenderer.render_shaded(balls)
    swap_and_save("obj_shaded")
```

```
circleMetaballsRenderer.render_color(balls)
swap_and_save("circle_color")

circleMetaballsRenderer.render_depth(balls)
swap_and_save("circle_depth")
```

The `swap_and_save` function swaps the back buffer to the front, and `save_buffer_as` reads the pixels from the new front buffer and saves the image to a file. The image is flipped vertically before saving, as OpenGL's coordinate origin is in the lower left of the image, returning rows of pixels bottom to top.

```
def save_buffer_as(filename):
    glReadBuffer(GL_FRONT)
    data = glReadPixels(0, 0, size[0], size[1],
                        GL_RGB, GL_UNSIGNED_BYTE)
    image = Image.frombytes("RGB", size, data)
    image = image.transpose(Image.FLIP_TOP_BOTTOM)
    image.save(filename)
```

5.2.2 Input images

In Section 4.1.1 the format of the input images to generate is described. How the images are generated is described in this section.

The input images are rendered using a billboard technique. A square quad is placed at the center of each metaball, facing the camera. The fragment shader draws a circle into each quad. UV coordinates within the fragment shader are relative coordinates representing the location of the fragment that is currently computed within the square quad, ranging from $(0, 0)$ to $(1, 1)$. The fragment is filled in when it is part of a circle with thickness 0.05, in uv coordinate space, around the center. Fragments that are not part of the circle are discarded, meaning that no color is written to the framebuffer at that location, and no depth is assigned. The metaballs are drawn one after the other. Depth buffering is enabled during this process, so that metaballs that a metaball that is in front of another metaball is drawn on top.

The position, color, and depth of the metaball are passed into the shader using uniform variables. Another uniform is used as the output type. The whole image is rendered twice, once with the circles drawn in the color of the corresponding metaball, and once with the color representing the depth of the metaball. The code of the fragment shader for the metaball quads is shown in listing 5.1.

```
uniform vec3 color;
uniform float radius;
uniform int outputType;

varying vec2 uv;
varying float depth;

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    if (length(fragCoord - 0.5) > 0.45
          && length(fragCoord - 0.5) < 0.5) {
        fragColor = vec4(color, 1.0);
        float centerDist = (0.5 - length(fragCoord - 0.5)) * 2;
        gl_FragDepth = depth;

        if (outputType == 1) {
            fragColor.rgb = vec3(gl_FragDepth);
        }
    } else {
        discard;
    }
}

void main() {
    mainImage(gl_FragColor, uv);
}
```

Listing 5.1: Circle fragment shader

5.2.3 Output images

For the target output images an accurate rendering of the metaballs is required. This was implemented using a ray-marching renderer (described conceptually in Section 3.1.4). The time it takes the target output images is not of concern, as in this offline step only test data is generated. The final rendering pipeline will not directly make use of this ray-marching renderer.

Instead of separate geometry for each metaball, like used for drawing the input images, the target output image is rendered entirely into a single full-screen quad. All the vertex shader does is pass on the uv-coordinates, which correspond to screen-coordinates from $(0, 0)$ to $(1, 1)$, to the fragment shader. The ray-marching algorithm is executed within the fragment shader. Position and color of the metaballs is passed to the fragment shader from the generation script via a uniform vector. A uniform is a piece of data that is passed to the fragment shader, which is constant for all fragments and doesn't depend

on varying data from the vertex shader. See code used for the target output fragment shader in listing 5.2.

```
#define METABALLS_COUNT 145

uniform vec4 metaballs[METABALLS_COUNT];
uniform vec3 metaballsColor[METABALLS_COUNT];
uniform int ballsCount;
uniform int outputType;

in vec2 uv;

#define FOV 30.
#define MIN_DEPTH 7.
#define MAX_DEPTH 25.
#define DEPTH_STEP 0.01
#define THRESHOLD 1.0

#define OUTPUT_TYPE_COLOR 0
#define OUTPUT_TYPE_DEPTH 1
#define OUTPUT_TYPE_NORMALS 2

// field function
float ff(vec4 metaball, vec3 pos) {
  vec3 metaballCenter = metaball.xyz;
  float metaballSize = metaball.w;
  float dist = length(pos - metaballCenter);
  return 1 / pow(dist / metaballSize, 2.);
}

// field function normal, gradient of the field function
vec3 ffn(vec4 metaball, vec3 X) {
    vec3 C = metaball.xyz;
    float metaballSize = metaball.w;
    return 2*(C - X) / pow(length(C - X), 4) * pow(metaballSize, 2);
}

// Returns the a point dist along a ray shot through the uv plane
// at coordinate uv
vec3 ray(vec2 uv, float dist) {
  float camPlaneDist = 1.0;
  float fov = radians(FOV);
  vec2 centeredUv = (uv - 0.5) * 2;
  vec3 dir = normalize(
```

```
      vec3(centeredUv * camPlaneDist * tan(fov), camPlaneDist));
  vec3 pos = dist * dir;
  return pos;
}

// Shoots a ray through the uv plane and returns a vector of
// (rgb, depth) of metaballs hit
vec4 depthRay(vec2 uv) {
  for (float depth = MIN_DEPTH; depth < MAX_DEPTH;
          depth += DEPTH_STEP) {
    vec3 pos = ray(uv, depth);
    float fieldValue = 0.;
    vec3 color = vec3(0.);

    for (int i = 0; i < ballsCount; i++) {
      float dist = length(pos - metaballs[i].xyz);
      float ballField = ff(metaballs[i], pos);
      fieldValue += ballField;

      color += metaballsColor[i]*ballField;
    }

    if (fieldValue > THRESHOLD) {
      vec3 normColor = color / fieldValue;
      return vec4(normColor, depth);
      break;
    }
  }

  return vec4(0., 0., 0., 1000.);
}

// Shoots a ray through the uv plane and returns the normal vector
// of the metaballs hit
vec3 normRay(vec2 uv) {
  for (float depth = MIN_DEPTH; depth < MAX_DEPTH;
          depth += DEPTH_STEP) {
    vec3 pos = ray(uv, depth);
    float fieldValue = 0.;
    vec3 norm = vec3(0.);

    for (int i = 0; i < ballsCount; i++) {
      float dist = length(pos - metaballs[i].xyz);
```

```
        float ballField = ff(metaballs[i], pos);
        fieldValue += ballField;

        norm += ffn(metaballs[i], pos);
      }

      if (fieldValue > THRESHOLD) {
        return normalize(norm);
      }
    }

    return vec3(0., 0., 0.);
}

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    fragColor = vec4(0.0, 0.0, 0.0, 1.0);

    if (outputType == OUTPUT_TYPE_COLOR) {
      vec4 depthRayValue = depthRay(uv);
      fragColor.rgb = depthRayValue.rgb;
    } else if (outputType == OUTPUT_TYPE_DEPTH) {
      vec4 depthRayValue = depthRay(uv);
      float depth = depthRayValue.w;
      fragColor.rgb =
          vec3((depth - MIN_DEPTH) / (MAX_DEPTH - MIN_DEPTH));
    } else if (outputType == OUTPUT_TYPE_NORMALS) {
      vec3 norm = normRay(uv);
      fragColor.rgb = (norm + 1.) * 0.5 * length(norm);
    }
}

void main() {
    mainImage(gl_FragColor, uv);
}
```

Listing 5.2: Metaball fragment shader

With a step size of 0.01 an accurate scene representation is obtained. A bigger step size leads to faster rendering, but also a less accurate result. If the step size is too big, artifacts are introduced and banding occurs.

The equation for the metaballs that was used is defined in the function `ff`, and is shown in 5.1. P is the center point of the metaballs and r is a size parameter for the metaball. In the fragment shader code, this size parameter is passed along in the center position vector as `metaball.w`.

$$f_{P,r}(X) = \frac{r^2}{(P-X)^2} \tag{5.1}$$

`ffn` is a function that returns the gradient of `ff` at a given point. This function is the derivative of the field function and is used to render the normal buffer. It is shown in mathematical notation in Equation 5.2.

There are 3 rendering modes: Color, depth, and normal buffer. These are the outputs discussed in Chapter 4.1.2 and can be set through a uniform variable passed into the shader. The depth value is scaled to move from the range of minimum to maximum depth to the range 0 to 1.

$$f_{P,r}(X) = 2r^2 \frac{(P-X)}{|P-X|^4} \tag{5.2}$$

5.2.4 Transformer

The images generated by the example generator in the previous section are not in the format that is ingested by the neural network during training. To simplify training, each example is encoded as a single image file, containing both the input and target images. A second step in the pipeline transforms the generated example images into this format.

To fit all eleven channels of information of both input and target output (three channels for input colors, one channel for input depth, three channels for target colors, one channel for target depth, and three channels for target normals) into a standard image format that the machine learning library Tensorflow is able to ingest, each example is encoded as a PNG image with alpha channel of dimension 1536×512 pixels. This width of 1536 fits three 512×512 pixel images side-by-side, giving the possibility to fit twelve 512×512 channels. One channel is left unused.

The first third of the image is used for the input, with the RGBA channels of the input mapped to the RGBA channels of the transformed example image. The second third is used for the RGBA channels of the target image. The final third of the image is used for the normals of the target and is stored in the RGB channels. The alpha channel is left unused.

To arrange the example images in that way, a short python script is used. It iterates over the output of the example generator, and reads the different images in using OpenCV. The numpy library is used to extract the relevant channels from the images and to concatenate them in the desired way. In the following listing the code of the transformer is shown.

```
white = np.full((IMAGE_WIDTH, IMAGE_HEIGHT), 255)

def read_image(path):
  img = cv2.imread(path, cv2.IMREAD_COLOR)
  img = cv2.resize(img, (IMAGE_WIDTH, IMAGE_HEIGHT),
          interpolation=cv2.INTER_CUBIC)
  return img


def create_ABs(examples, output_dir):
  for in_idx, dir in enumerate(examples):
    circle_depth_path = os.path.join(data_dir, dir, "circle_depth.png")
    circle_color_path = os.path.join(data_dir, dir, "circle_color.png")
    obj_depth_path = os.path.join(data_dir, dir, "obj_depth.png")
    obj_color_path = os.path.join(data_dir, dir, "obj_color.png")
    obj_normals_path = os.path.join(data_dir, dir, "obj_norm.png")

    circle_depth = read_image(circle_depth_path)[:,:,0]
    circle_color = read_image(circle_color_path)
    obj_depth = read_image(obj_depth_path)[:,:,0]
    obj_color = read_image(obj_color_path)
    obj_normals = read_image(obj_normals_path)

    rgb = np.concatenate((circle_color, obj_color, obj_normals),
                         axis=1)
    alpha = np.concatenate((circle_depth, obj_depth, white),
                           axis=1)
    alpha = np.expand_dims(alpha, 2)

    img = np.concatenate((rgb, alpha), axis=2)

    out_path = os.path.join(output_dir, dir + '.png')
    cv2.imwrite(out_path, img)
```

Listing 5.3: Example transformer script

5.3 Network architecture

The network architecture was described conceptually in Section 4.2. In this section the layers and their parameters will be described in more detail.

Tensorflow uses a dataflow graph to represent computation within a network, as well as operations that mutate the network's state. All data that flows through the graph is represented in a datastructure called a tensor. A tensor is a generalization of vectors and

matrices to potentially higher dimensions, or an n-dimensional array. A dataflow graph can be composed flexibly from primitive nodes, such as variables and fixed values, and basic operations such as addition and multiplication. Higher level nodes can be built on top of those basic nodes, and Tensorflow comes with a lot of typical higher level objects built-in. These include deep learning concepts such as nodes for convolutional layers and ReLUs.

In the following sections, the different parts of the network are described in detail. The network architecture is adapted from the work of Isola et al. on pix2pix [IZZE17]. They originally used Torch, another machine learning library, to implement their network. Code for this thesis is based on a Tensorflow port of pix2pix[2]. The final network has a parameter count of 24 706 760 parameters that have to be trained. It was trained and run in GPU mode on a Nvidia GeForce GTX 1080, for which the Nvidia libraries CUDA and CuDNN are required.

5.3.1 Generator

Layer name	Output tensor size	Dropout probability
input	$512 \times 512 \times 4$	0
encoder_1	$256 \times 256 \times 32$	0
encoder_2	$128 \times 128 \times 64$	0
encoder_3	$64 \times 64 \times 128$	0
encoder_4	$32 \times 32 \times 256$	0
encoder_5	$16 \times 16 \times 256$	0
encoder_6	$8 \times 8 \times 256$	0
encoder_7	$4 \times 4 \times 256$	0
encoder_8	$2 \times 2 \times 256$	0
decoder_8	$4 \times 4 \times 512$	0.5
decoder_7	$8 \times 8 \times 512$	0.5
decoder_6	$16 \times 16 \times 512$	0.5
decoder_5	$32 \times 32 \times 512$	0
decoder_4	$64 \times 64 \times 256$	0
decoder_3	$128 \times 128 \times 128$	0
decoder_2	$256 \times 256 \times 64$	0
decoder_1	$512 \times 512 \times 7$	0

Table 5.2: Parameters for generator layers.

[2] https://github.com/affinelayer/pix2pix-tensorflow

The convolutional and deconvolutional layers used for the generator are listed in Table 5.2. The table shows the name of the layer and its output size, as well as the dropout probability used in some layers. Dropout is a technique to prevent overfitting, in which random units are dropped according to some probability [SHK+14]. Some details such as ReLU layers between the convolutional layers are not shown in the table but will be described below.

At the start of the generator is the input layer, that is the input image that will be transformed into another image as it passes through the network. It has four channels containing the RGB color and depth information of the input. The generator network is U-shaped with skip connections. As the data flows through the first half of the generator, the width and height of the output tensor get smaller, but the tensor gets deeper with more channels. Afterwards it passes through a series of deconvolutional layers, increasing the tensor width and height again, but reducing the depth. The final layer at the end is back to the original input size, but contains the seven channels used for RGB color information, depth, and normals, as required by the output.

The layer names come from the generator's relation to autoencoders. The "encoder" layers are convolutional layers, implemented using Tensorflow's `conv2d` layer, created using the Tensorflow Python API:

```
tf.layers.conv2d(batch_input, out_channels, kernel_size=4,
    strides=(2, 2), padding="same", kernel_initializer=initializer)
```

The `batch_input` variable is the tensor to convolve over, for the first layer this is the input itself. The number of out channels is shown in Table 5.2 above. A stride of 2 and a kernel size of 4 is used. The `kernel_initializer` parameter determines the way the filter kernels are initialized before training has started. We use a random initializer `tf.random_normal_initializer(0, 0.02)` that samples values from a normal distribution with mean 0 and standard deviation of 0.02. After each convolutional layer, a leaky ReLU layer is inserted for non-linearity, using a slope of 0.2 in the negative range.

The "decoder" layers use a deconvolutional layer, or strided convolution, and output tensors that have bigger width and height than their input. Skip connections added here allow the deconvolution to also get data directly from the corresponding encoder layer. These skip connections are implemented by concatenating the output from the layer before the decoder layer with the output from the corresponding encoder layer along the depth axis, using the `tf.concat` function:

```
tf.concat([previous_layer, skip_layer], axis=3)
```

The resulting tensor has the same width and height as the output of the layers it concatenates, but combines the depth channels of both layers. Before it is passed as input to the deconvolution layer, a ReLU layer is inserted using the function `tf.nn.relu`. The deconvolution is implemented using Tensorflow's `conv2d_transpose` function:

```
tf.layers.conv2d_transpose(batch_input, out_channels,
```

```
kernel_size=4, strides=(2, 2), padding="same",
kernel_initializer=initializer)
```

Transposed convolution is another name used for deconvolution. The number of output channels for each layer is shown in Table 5.2 above. For the `kernel_size`, `strides`, `padding` and `kernel_initializer` parameters the same values as for the `conv2d` layers is chosen. Dropout is added after the transposed convolution layer using the function `tf.nn.dropout(output, keep_prob=1 - dropout)`.

The last decoder layer, "decoder_1", has a skip connection to the first encoder layer. After deconvolution, the function `tf.tanh` is applied to the output as activation function. This is the final output of the generator, having the same width and height as the input, and containing 7 channels for RGB color, depth, and normal vectors.

5.3.2 Discriminator

Layer name	Output tensor size
input	$512 \times 512 \times 11$
layer_1	$254 \times 254 \times 64$
layer_2	$125 \times 125 \times 128$
layer_3	$60 \times 60 \times 256$
layer_4	$55 \times 55 \times 512$
layer_5	$50 \times 50 \times 1$

Table 5.3: Parameters for discriminator layers.

Next, we will discuss the structure of the discriminator. Table 5.3 gives an overview of the layers the discriminator consists of.

The input layer for the discriminator has the width and height of our input and target images, and has a depth of 11 channels. These 11 channels are the concatenation of the 4 channels of the input image, used as the condition for the discriminator, and 7 channels of the image to classify as correct or incorrect. That image may come from the target images, as rendered by the ray marching renderer, or is the result of the generator part of the network.

The layers are convolutional layers, using the `conv2d` layer implementation, with a kernel size of 8. Most layers use a stride of 2, except for the last layer with a stride of 1.

After each convolutional layer follows a leaky ReLU, with a slope of 0.2 in the negative range. Only for the last layer is the ReLU replaced with a sigmoid layer instead. How the output of the discriminator, a 50×50 tensor, is used during training is discussed in Section 5.3.4.

5.3.3 Input layers

The input layer of the network looks different, depending on if the network is being trained or used for inference.

Training input layer

In training mode, the examples the network is trained on are read from the file system. They are stored by the transformer, as discussed in Section 5.2.4, as alpha PNG files with a dimension of 1536×512 pixels.

All example files are listed as paths in a list `input_paths`, and a path queue is created using Tensorflow's string input producer:

```
path_queue = tf.train.string_input_producer(input_paths,
                shuffle=a.mode == "train")
```

These paths are passed to a `WholeFileReader` node, decoded using a PNG decoder node, then converted into a float representation:

```
reader = tf.WholeFileReader()
paths, contents = reader.read(path_queue)
raw_input = tf.image.decode_png(contents, channels=4)
raw_input = tf.image.convert_image_dtype(raw_input, dtype=tf.float32)
raw_input.set_shape([None, None, 4])
```

Some tensor slicing is performed to extract parts of the input to separate variables. Tensorflow supports a syntax for slicing that is very similar to that of numpy:

```
# break apart image pair and move to range [-1, 1]
width = tf.shape(raw_input)[1]  # [height, width, channels]

part_width = width // 3
circles = raw_input[:, 0:part_width, :]
obj = raw_input[:, part_width:2 * part_width, :]
obj_norm = raw_input[:, 2 * part_width:, 0:3]
obj_norm.set_shape([None, None, 3])
```

Finally the output is reassembled into a single $512 \times 512 \times 7$ tensor:

```
inputs = preprocess(circles)
targets = preprocess(tf.concat([obj, obj_norm], 2))
```

The `preprocess` just moves the values of the image from the range $[0, 1]$ to the range $[-1, 1]$. This is all the preparation done on the examples, it is then passed as input to the generator and discriminator networks.

Inference input layer

Because we want the network to be usable in a real-time interactive setting, writing the input out as PNG encoded image file, only to be read back in would be inefficient. Instead, we want to be able to pass already decoded image data from main memory to the network. To do this, the input layer of the network has to be swapped out during inference.

Tensorflow has the concept of placeholders, tensors that can be supplied with data that are passed when running a feed-forward pass of the network. Placeholders have to be named, so that data is supplied to the correct tensor. We will have images in `uint8` encoding with a range of $[0, 255]$ in main memory, but will again transform them to float tensors with a range of $[-1, 1]$.

```
input_placeholder = tf.placeholder(tf.uint8,
    shape=[CROP_SIZE, CROP_SIZE, 4], name='input_placeholder')
scaled_input_placeholder = tf.cast(input_placeholder, tf.float32)
scaled_input_placeholder = tf.divide(scaled_input_placeholder, 255.0)
inputs = preprocess(scaled_input_placeholder)
```

The targets tensor is not used during inference, however a tensor of the correct size still has to be supplied for the network to be consistent. For this, we just create a tensor filled with ones:

```
targets = tf.ones(shape=[CROP_SIZE, CROP_SIZE, 7],
    dtype=tf.float32, name='no-target')
```

5.3.4 Loss and update layers

The final part of the network design to discuss is how it performs its training operations. First, two copies of the discriminator are created, one that will receive "real" pairs and one that will receive "fake" pairs of input and output images. "Real" images in this case refers to the target images coming from the ray-marching renderer, and "fake" images are the images produced by the generator. These two discriminators are not independent however, they share their underlying parameters:

```
with tf.name_scope("real_discriminator"):
    with tf.variable_scope("discriminator"):
        predict_real = create_discriminator(inputs, targets)

with tf.name_scope("fake_discriminator"):
    with tf.variable_scope("discriminator", reuse=True):
        predict_fake = create_discriminator(inputs, outputs)
```

As discussed in Section 5.3.2, the discriminator outputs a $50 \times 50 \times 1$ tensor. From this tensor we calculate the discriminator loss by calculating the mean of the log of difference to the target values:

```
with tf.name_scope("discriminator_loss"):
    # minimizing -tf.log will try to get inputs to 1
    # predict_real => 1
    # predict_fake => 0
    discrim_loss = tf.reduce_mean(
      -(tf.log(predict_real + EPS) + tf.log(1 - predict_fake + EPS)))
```

The loss for the generator is calculated as the weighted sum of two terms, GAN loss and L1 loss. GAN loss is calculated similarly to the discriminator loss, as measure of how well the generator is able to fool the discriminator. The L1 loss is based on the mean of the pixelwise distance between the output and target values. The GAN and L1 loss have been assigned weights of 500 and 0.5 respectively.

```
with tf.name_scope("generator_loss"):
    # predict_fake => 1
    # abs(targets - outputs) => 0
    gen_loss_GAN = tf.reduce_mean(-tf.log(predict_fake + EPS))
    gen_loss_L1 = tf.reduce_mean(tf.abs(targets - outputs))
    gen_loss = gen_loss_GAN * a.gan_weight + gen_loss_L1 * a.l1_weight
```

On each step during training, first the discriminator and then the generator is updated. The Adam optimizer [KB14] is used, with a learning rate of 0.0002, and momentum parameters $\beta_1 = 0.5$, $\beta_2 = 0.999$. In the following listing, the training operations for both discriminator and generator are defined, and then grouped together to be able to be applied together during training. These operations will be excluded at inference time, as updating the network is not desired at that time.

```
with tf.name_scope("discriminator_train"):
    discrim_tvars = [var for var in tf.trainable_variables()
        if var.name.startswith("discriminator")]
    discrim_optim = tf.train.AdamOptimizer(a.lr, a.beta1)
    discrim_grads_and_vars = discrim_optim.compute_gradients(
      discrim_loss, var_list=discrim_tvars)
    discrim_train =
      discrim_optim.apply_gradients(discrim_grads_and_vars)

with tf.name_scope("generator_train"):
    with tf.control_dependencies([discrim_train]):
        gen_tvars = [var for var in tf.trainable_variables()
            if var.name.startswith("generator")]
        gen_optim = tf.train.AdamOptimizer(a.lr, a.beta1)
        gen_grads_and_vars = gen_optim.compute_gradients(gen_loss,
            var_list=gen_tvars)
        gen_train = gen_optim.apply_gradients(gen_grads_and_vars)
```

```
ema = tf.train.ExponentialMovingAverage(decay=0.99)
update_losses = ema.apply([discrim_loss, gen_loss_GAN, gen_loss_L1])

train=tf.group(update_losses, gen_train)
```

5.4 Demo renderer

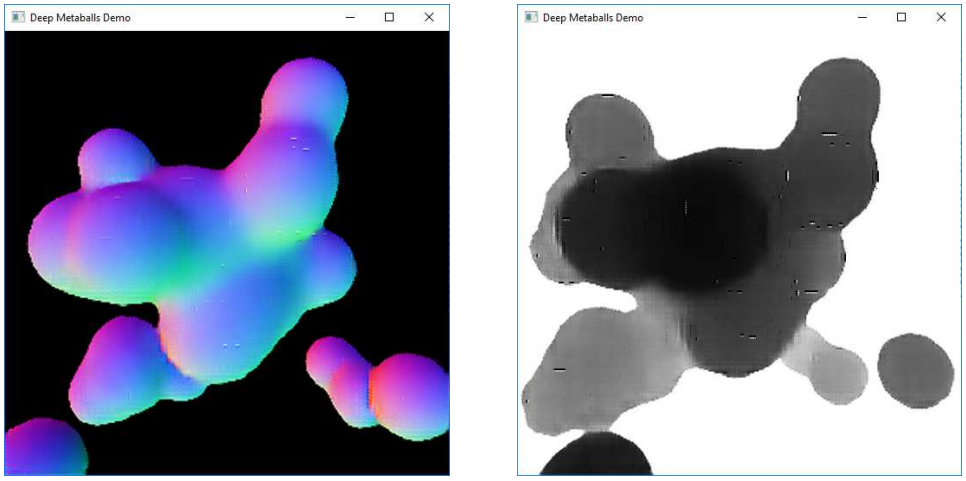


Figure 5.1: Screenshots of the demo renderer.

A demo application that uses the neural network developed in this thesis to render metaballs was also implemented. This realtime renderer is interactive, it allows the user to navigate a scene of metaballs using keyboard and mouse. Figure 5.1 shows a scene in the demo renderer, with different viewpoints and rendering modes.

The demo application serves as a proof of concept of using the neural renderer for interactive applications. It also serves as a way to evaluate the network interactively. The source for the demo application is available in the source repository as well.

A scene with randomly placed and colored metaballs is generated by the demo application, which can be navigated by the user using the WASD keys to move the camera, as well as by dragging the mouse to change the camera angles for different perspectives on the scene. Different rendering modes allow the user to see the final output of the neural renderer, as well as the input given to the neural network (as described in 4.1.1) and a more accurate version using a raymarching renderer, as used to generate the target images for training the network. For the different renderers (input, output, and raymarched target) the buffer to show can also be selected amongst color, depth, and normal buffers. The user can switch between these rendering modes by pressing a number key.

The FreeGLUT library (an open source version of the OpenGL Utility Toolkit) was used to set up the window with its associated OpenGL context and to handle input events.

The window is set to be double-buffered, so GLUT sets up two OpenGL buffers. One buffer, called the front buffer, is to be shown on screen. The second buffer is called the back buffer, and this is where drawing commands are issued to. After drawing a frame into the back buffer is completed, the buffers get swapped - the back buffer becomes the front buffer, and thus its content gets shown on screen. Using a back buffer prevents incomplete images to be shown on screen during drawing.

The process for rendering a frame in the demo application is similar to the process used in training. With the data about position and color of metaballs that is generated for the scene, the simple circle input is rendered into the back buffer, reusing the same shader as during training. The image data is then read back into main memory using the glReadPixels function, and fed as data to the model. The dataflow graph of the neural network is adapted slightly compared to the graph used during training. When feeding the data back into the model it is already in decoded form, an RGBA image represented as $512 \times 512 \times 4$ unsigned byte array, so the node used to decode PNG images is skipped to skip unnecessary processing. The model is then run, and the result is extracted from the output node. It is then uploaded as a texture to the GPU and drawn as a texture onto a quad covering the screen. This back-and-forth copying between CPU and GPU could probably be avoided for better performance.

CHAPTER 6

Results

In this chapter, we show the results. First, we will discuss the hardware this was run on, then show examples of results, performance of the network, and lastly see the training progress.

All the data that was generated, including the training input and target output images, as well as the fully trained model, can be found online at `https://www.gitlab.com/robhor/deep-metaballs`.

6.1 Hardware

Table 6.1 shows an overview of the hardware that the network was trained and run on.

CPU	Intel(R) Core(TM) i7-6700 CPU @ 4.00 GHz
Memory	16 GB RAM
Storage	Crucial MX300 CT525MX300SSD1 525 GB SSD
Graphics cards	Nvidia GeForce GTX 1080 Nvidia GeFore GTX Titan

Table 6.1: Hardware of the PC the network was trained and run on.

6.2 Result examples

In Figure 6.1 an example of the output of the neural network is shown. The output of the neural network is on the left, and the target output, rendered with a ray-marching renderer, is on the right. From top to bottom, the image shows the color, normals, and depth buffer respectively. All images were generated with a size of 512×512 pixels. Figure 6.2 shows other samples of the network output. Discussion of the results are found in Chapter 7.

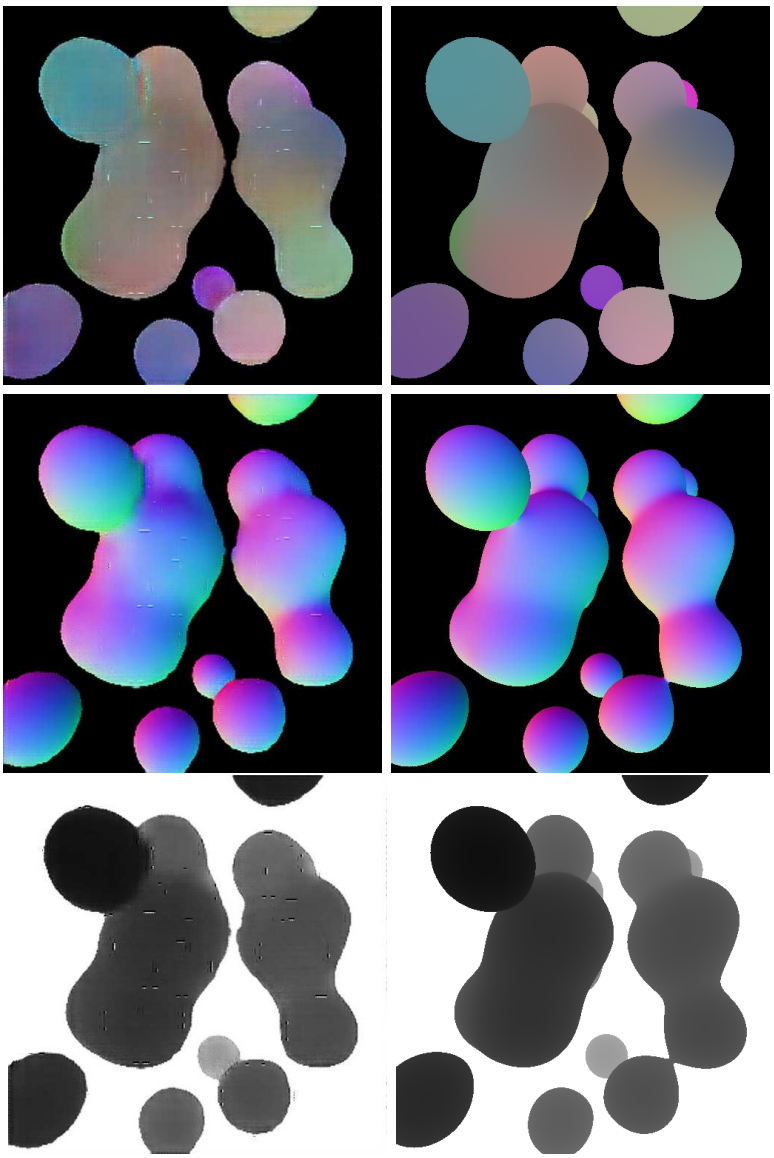


Figure 6.1: Result example. The output of the neural network is on the left, the target output is on the right. From top to bottom: color, normals, and depth buffer.

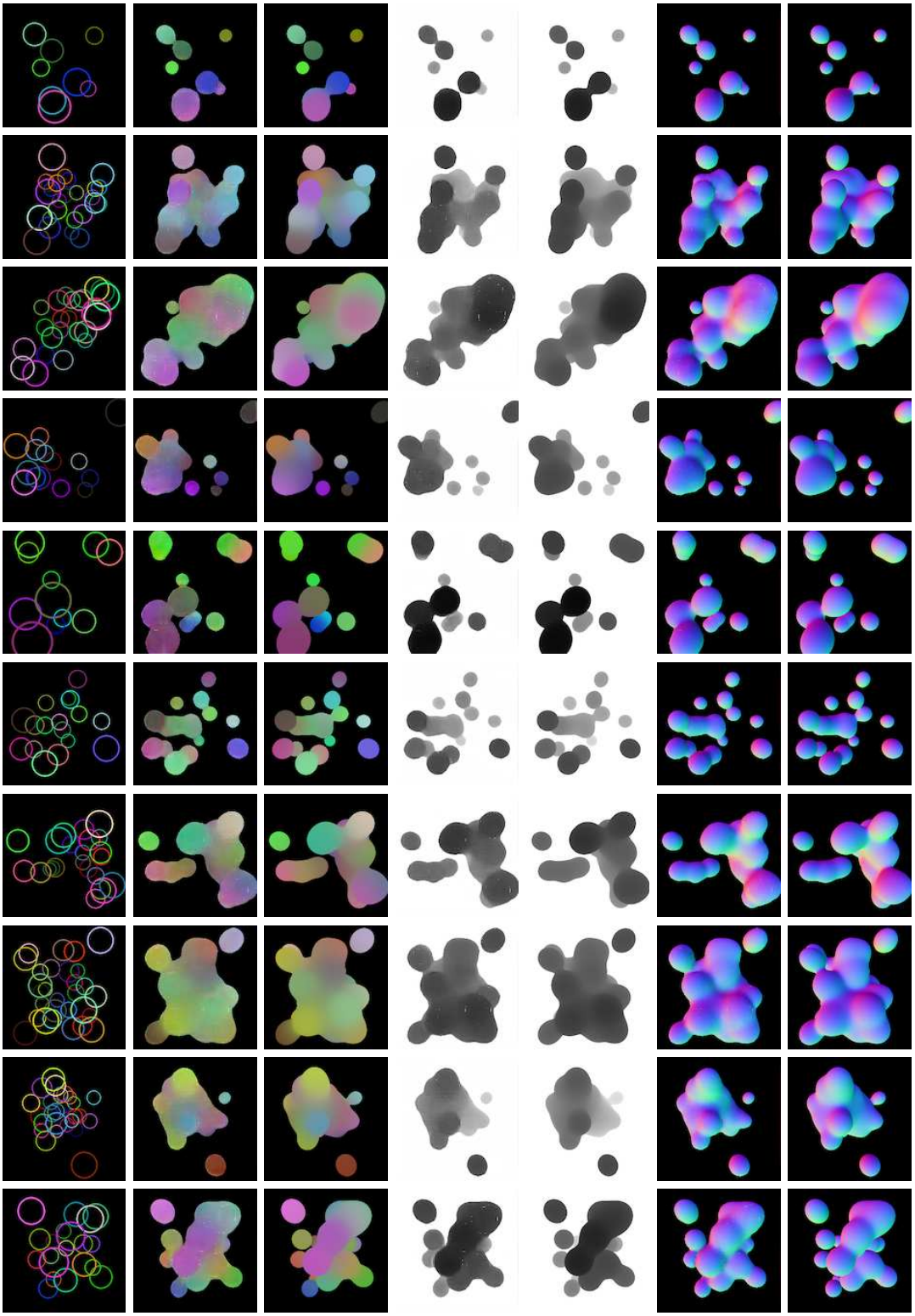


Figure 6.2: Result examples. The first column depicts the input, the following columns depict the color, depth, and normal buffers of the neural network output and target output.

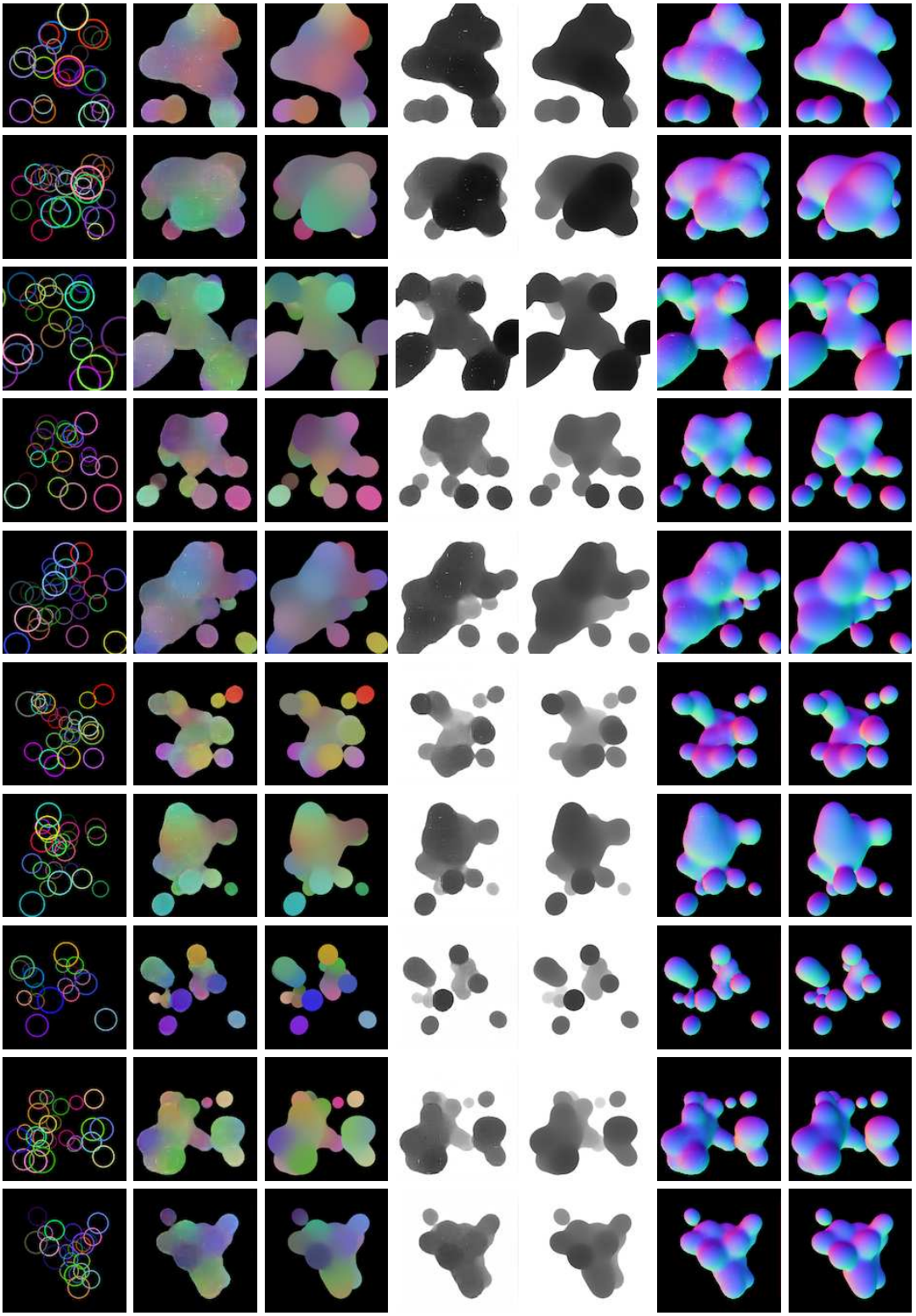


Figure 6.2: Result examples. The first column depicts the input, the following columns depict the color, depth, and normal buffers of the neural network output and target output.

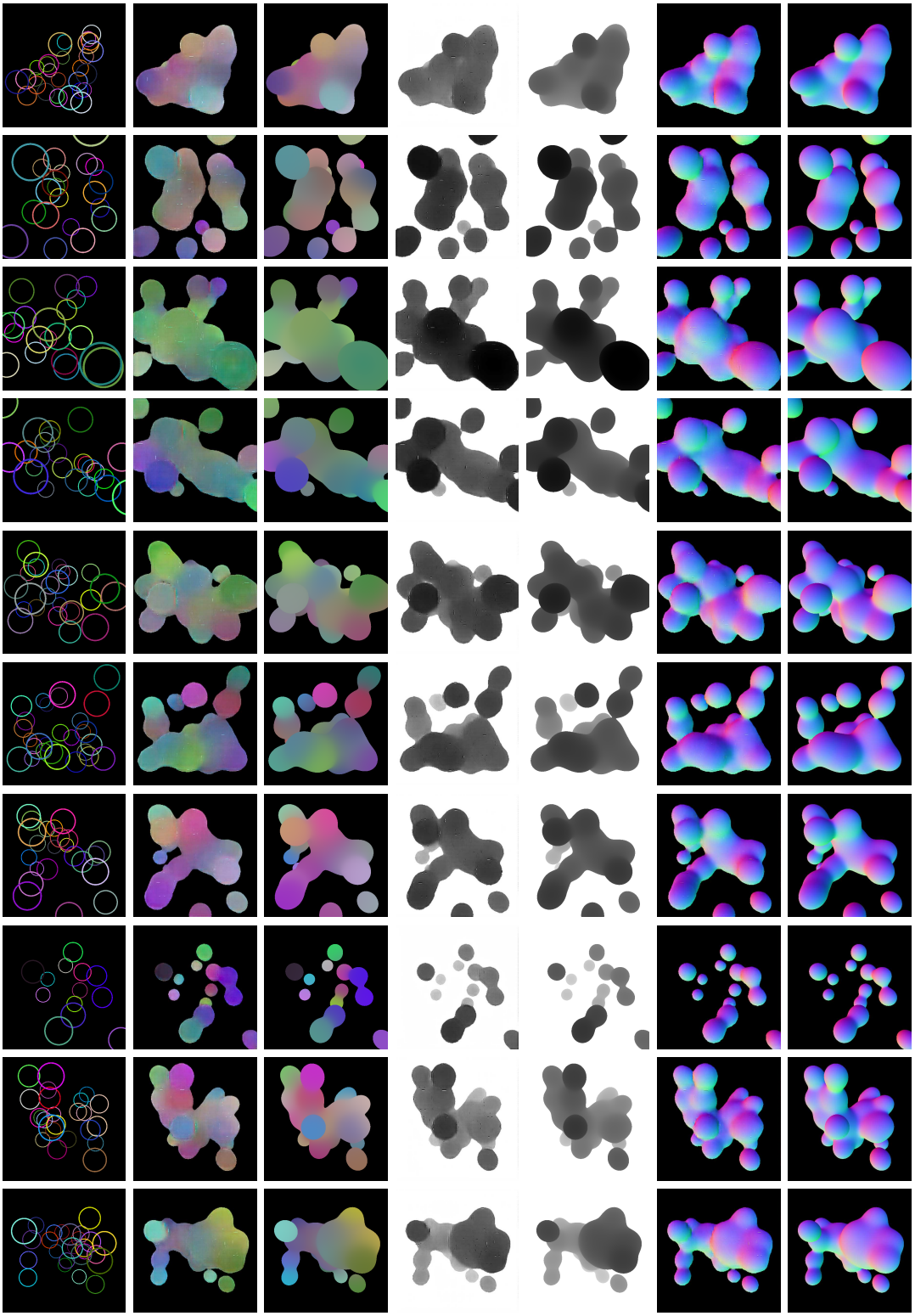


Figure 6.2: Result examples. The first column depicts the input, the following columns depict the color, depth, and normal buffers of the neural network output and target output.

6.3 Performance

Performance of the neural renderer was measured in the demo application (see Section 5.4). The numbers can be seen in Table 6.2.

Time to render neural network input image	$<$ 1ms
Time to read input image back in to main memory	2ms
Time to run neural network on input	18ms
Time to render finished frame	2ms
Total time	22ms

Table 6.2: Performance numbers.

Rendering the input image takes less than a millisecond. Reading the image back into main memory takes 2 milliseconds. The neural network takes on average 18 milliseconds to run on our hardware. So the finished frame takes about 22 milliseconds to render. Using that we can achieve an actual frame rate of 45 frames per second. Although some inefficiencies remain in the demo application, and a speedup to 50 frames per second without change to the neural network is doable.

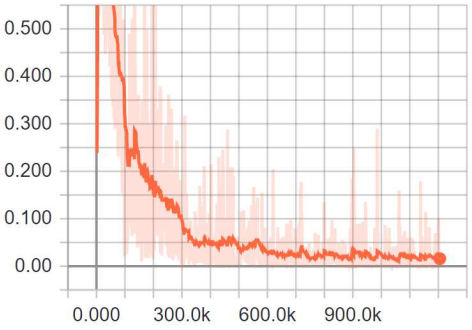
6.4 Training progress

Figure 6.3 depicts several graphs, as rendered by Tensorboard, of the loss values for the different parts of the network. The losses and how they are calculated are discussed in Section 5.3.4. A smoothed line is shown over the noisy raw data points to show the general trend over many iterations. The horizontal axis in all the graphs is the training iteration, and the vertical axis is the loss value.

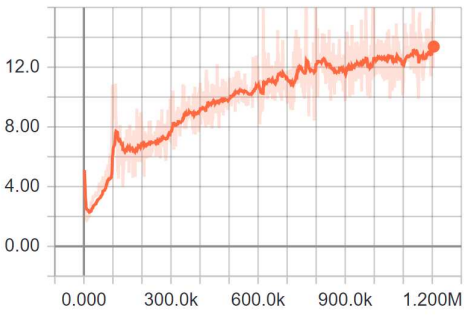
Discriminator loss fell quickly over many iterations, as did the generator L1 loss. Progress for both these loss values slowed down and stabilized after about 400 thousand iterations.

Generator GAN loss is also shown. It is rising steadily after slowing down after about 900 thousand iterations. The generator GAN loss has a small weight compared to the L1 loss, so the network is less inclined to optimize for smaller GAN loss.

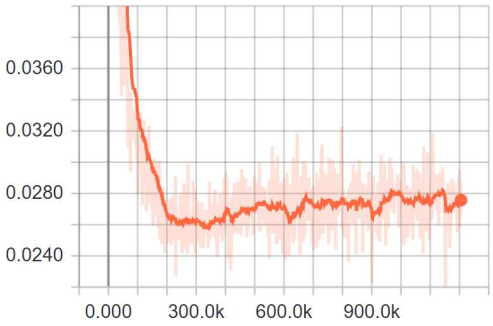
In Figure 6.4 the output of the network after different numbers of training iterations is shown. Training on this data set of 20000 examples is fast. On an Nvidia GeForce GTX 1080 graphics card, 1000 training iterations were completed after just 4 minutes. The results improve visibly quickly at first, but further progress is taking increasingly long. Iteration 30000 was finished after approximately 2 hours.



(a) Discriminator loss



(b) Generator GAN loss



(c) Generator L1 loss

Figure 6.3: Discriminator and generator loss over time.

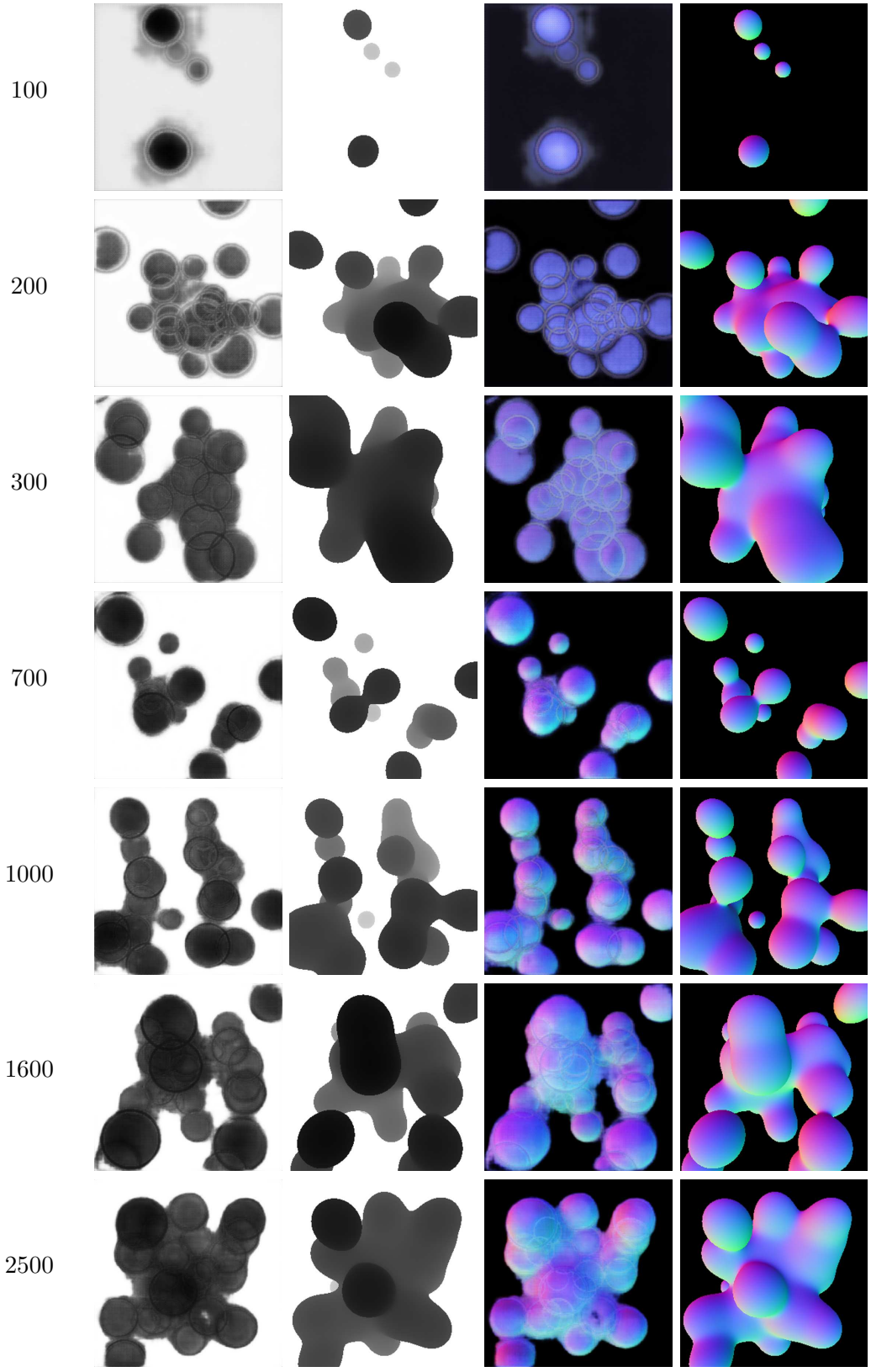


Figure 6.4: Training progress. From left to right: Iteration number, output depth, target depth, output normals, target normals

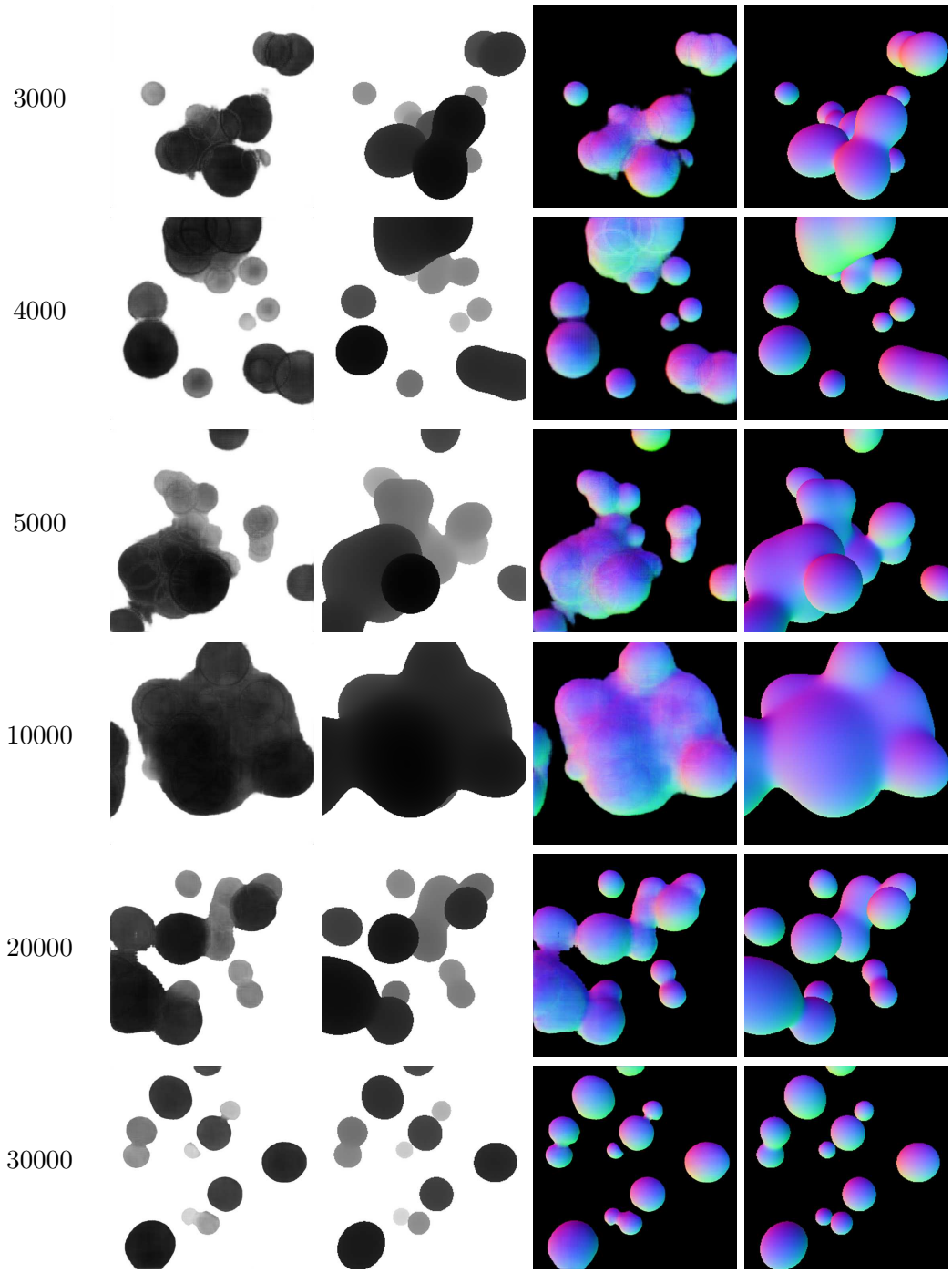


Figure 6.4: Training progress. From left to right: Iteration number, output depth, target depth, output normals, target normals

CHAPTER 7

Critical Review

In this chapter we take a closer look at the approach taken in this work, and will evaluate the results. We will discuss what worked well and what did not work so well, and how it could be improved in the future.

For the discussion of the results, we will take the result shown in Figure 7.1 as an example. We will discuss several aspects of the result image quality, including the general shape of the rendered object, and how closely the color, depth and normal buffers match the target output. To help in the analysis of the results, we created some visualizations of the differences between aspects of the output of the neural network and the target output. These visualizations are shown in Figure 7.2. There, the top left image shows a comparison of outlines, where the shape of the target output is in the red channel and the shape of the neural network output is in the green channel. On the top right, bottom left and bottom right we show the absolute difference between the neural network output and the target output in the color, depth, and normal buffers, respectively. The absolute difference has been doubled for better visibility, as otherwise the difference is hard to see.

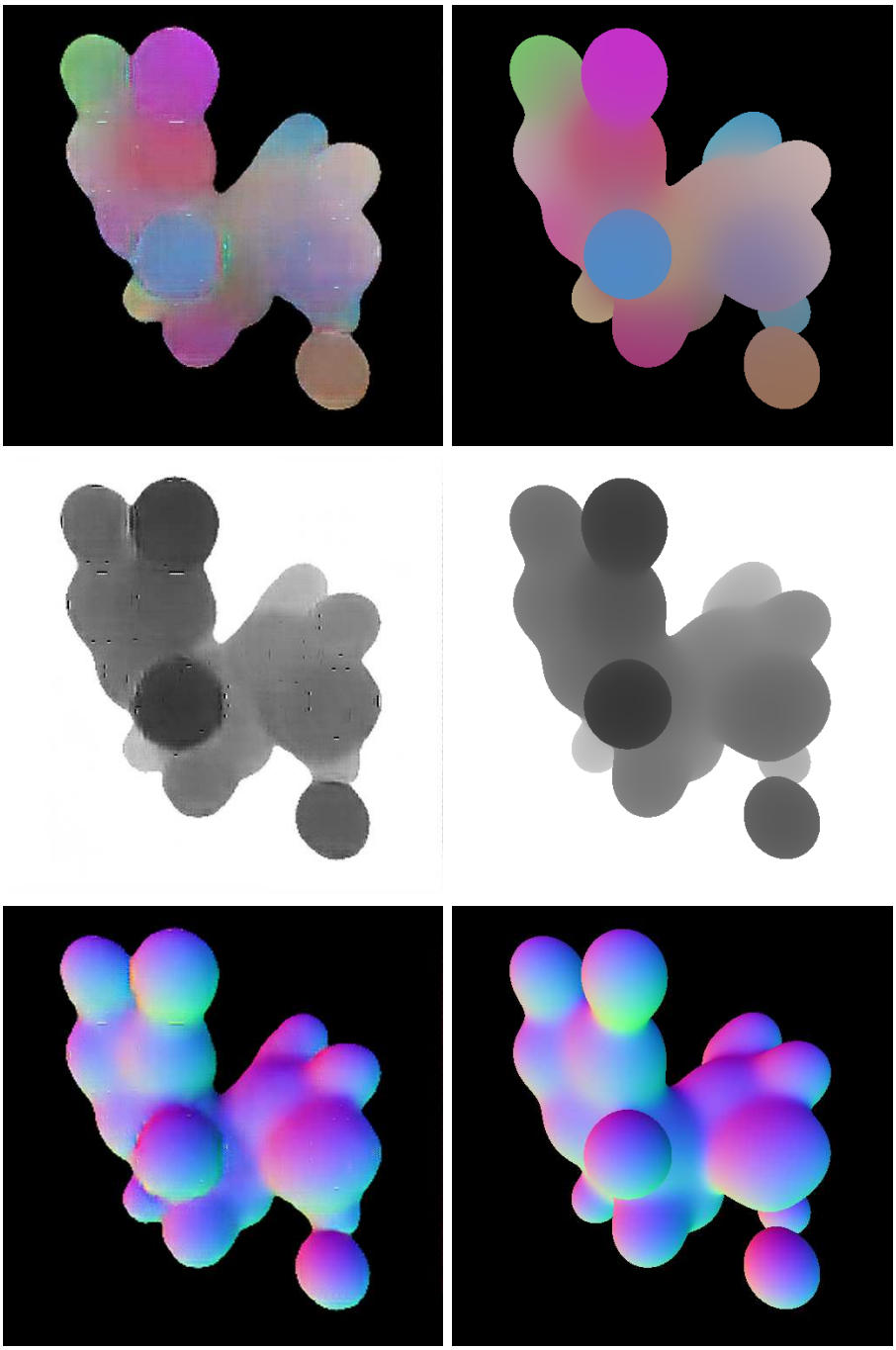


Figure 7.1: Example metaball scene with our result on the left, target output on the right.

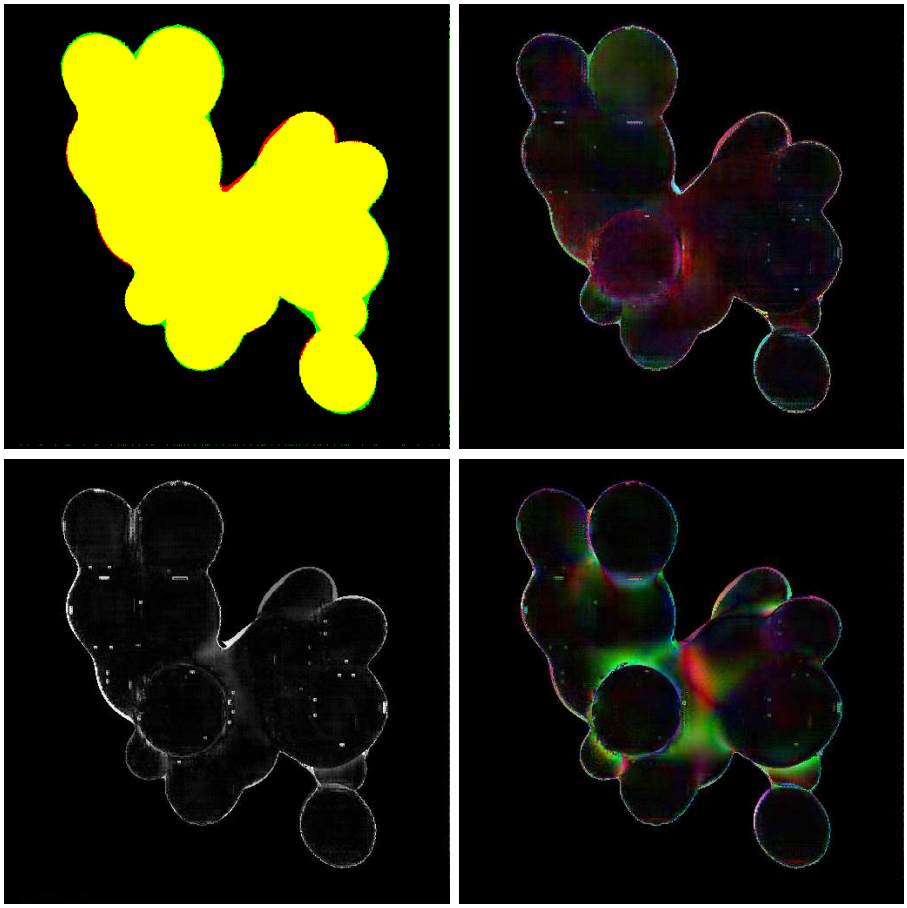


Figure 7.2: Top left: Comparison of outlines (yellow is overlap, red is target, green is our result).
Top right: Enhanced difference in color buffer.
Bottom left: Enhanced difference in depth buffer.
Bottom right: Enhanced difference in normal buffer.

7.1 Output images

On the top left of Figure 7.2 a comparison between the shapes of our result and the target output, as obtained by the ray-marching renderer on the same input, is depicted. To create this image, the red channel of the normal buffer of both the neural network output and the target output was taken and thresholded, so that everything that is not black, and thus part of the background, got a value of white. This produced a binary black-and-white image, with the shape of the metaball object in white. These thresholded images have then been combined into a single image, with the shape of the target output in the red channel and the shape of the neural network output in the green channel. Everywhere there is yellow in the combined image, shown in Figure 7.2, the shapes of both rendering methods overlap, so are in agreement. The red parts on the edges of the shape are where the target output produced pixels that are part of the object but where the neural network renderer didn't render an object. Finally, the green parts are where the neural network produced pixels as part of the object that should not be there according to the ray-marching renderer.

Most of the shape outline matches well, but not perfect. In general, the neural network output tends to be blurrier. Its edges are fuzzier and sometimes it creates a smoother blended outline between metaballs that should not be merged. An example of this is seen in the lower right of the example output, where a metaball is behind the foreground object, and should not be merged into the main object. In the target output, there exist clean lines and the farther metaball looks spherical, whereas the neural network output produces a smooth transition with the metaball that overlaps it on the top. Note that the unwanted transition is not as pronounced with the metaball that overlaps it on the bottom, that is also closer to the camera. There, the depth-edge has been preserved better. In other examples, bridges and transitions between metaballs that exist in the target output have not been reproduced in the neural network output.

One factor that influences how the target output shape might not always agree with the neural network output shape is the use of a field function without finite support. In the ray-marching renderer, metaballs from the entire scene have influence at every point in space, even if only a small influence. If there is a densely packed cluster of metaballs, these influences add up and push the outline of the resulting object farther out. The neural network only takes patches of input into account, which is better for modeling the behavior of field functions with finite support.

The output of the neural network renderer contains big artifacts that can be seen well in the bottom left of Figure 7.2. The location of these artifacts have been found to correspond to the edges of the circles in the neural network input. This suggests that another input structure could be tried, or to reduce the strength of some of the skip connections that link the input directly to the image generation part of the network.

Another issue of the output is that its surfaces are not as smooth as the surfaces in the target image. Dust-like artifacts are seen on a small regular grid on the surface. These artifacts might be related to the chosen convolution kernel sizes and strides. However,

these artifacts were not noticed in earlier attempts where the network was only trained on and only generated the depth buffer.

The color output of the neural network matches the target output well in general, but it is not as smooth as the target output. In places where the neural network decided to blend metaballs together, when there should be a hard edge between them, the color between the metaballs was also blended together incorrectly. This can also be seen in the top right image in Figure 7.2, where the largest errors occur at the edges of incorrectly blended metaballs (and where the outline of the shape does not match).

The biggest issues in the depth output of the neural network are the artifacts and in places where hard edges should exist between metaballs at different distances that should not be blended together. Overall, the error in the depth output is generally low, as seen in the bottom left of Figure 7.2.

The normals on the neural network output look convincing in general. But again, the biggest error exists where metaballs at different distances have been blended together incorrectly. That error can be seen in Figure 7.2, where metaballs that have been merged incorrectly horizontally are showing up in red, and those merged incorrectly vertically are showing up in green, as the red and green channels correspond to the x and y components of the normal vector.

Overall, the biggest source of issues in the neural network output was that it blended metaballs together, even if they were separated by a large distance in depth. This caused colors and depth to be merged and the output normals to be wrong accordingly.

One potential way to deal with this issue would be to give the depth differences higher importance during training, by creating a loss value that measures the difference in depth values between the generator output and the target output, and weighing that difference higher than before. Another idea is to use a different depth representation. Instead of the linear depth scale used for this work, a quadratic or logarithmic scale that assigns bigger differences between near depth values than between far depth values might be useful. In the far distance, depth differences cannot be seen so easily, so reserving more space for the near values could be beneficial.

7.2 Viewpoint stability

In the interactive demo renderer another negative effect that is not visible in the comparison images can be observed. Small differences in the viewpoint or viewing angle can cause slight differences in the produced shape. Sometimes, bridges between metaballs are created and broken. This can be jarring when having a dynamic scene, but is not surprising since the network was only trained on single images of static scenes.

A way to keep the produced image more stable between frames might be the use of a recurrent neural network (RNN) or a long short-term memory network (LSTM). These types of neural networks are specifically designed to handle sequences, such as sequences

of frames in videos. They can take data from several previous frames of the sequence as input for the next frame. That way, outline information may stay more consistent across multiple frames and changes could be smoothed out, producing a less jarring image sequence.

7.3 Input encoding

In this thesis, an image-to-image translation approach was used as the model for our network. Given metaball data, including position and color of the metaballs, that required an artificial input image to first be synthesized and rendered, as the input to the network had to be an image. Therefore, the question arises if that step could be skipped by using the metaball data more directly as input for the neural network. However, convolutional neural networks, that have proven very successful in image-related tasks, have achieved their performance by making use of the locality of data in images. A simple list of vectors would not have the same locality. The input image we used can be interpreted as a 2D matrix containing information about the metaballs at each entry. But the encoding of the data therein could look different. A 3D volumetric input data structure could be used, which might provide a better, more flexible way to represent depth information, but the problem of converting a list of metaballs to that data structure would arise. And by using a 2D representation that already approximates the perspective projection the network has to learn less.

7.4 Performance

With approximately 45 frames per second the demo renderer is usable for interactive frame rates, but these frame rates are not ideal. Increasing resolution comes at some computational cost, but convolutions, which form the main operations of the neural network, are highly parallelizable. Graphics cards are well equipped to perform this kind of task. There is optimization potential in the demo renderer, in which data is copied from GPU memory to CPU memory, only to copy it back to GPU memory for Tensorflow to process.

An interesting property of the neural network approach is that it scales well with number of metaballs in the scene. Rendering the circle input image is very fast, and the neural network processing takes the same computation steps regardless of the number of circles in the image.

7.5 Conclusion

The neural network learned to reproduce many aspects of the target images very well, and the outputs are blobby objects with similar structure. Overall, the output image tends to be blurry though, and metaballs separated by distance in depth are too easily blended by the network. With adjustments to the input data structure and to how the

loss values are calculated, improvements to these issues can be expected. However, this renderer might be too blurry for detailed rendering of close-up metaballs, so a hybrid renderer approach that uses this renderer for metaballs that are farther away from the camera, thus require less visual accuracy, can be considered. The inclusion of the depth information makes the output of the renderer work well with rendering pipelines using deferred shading and can be composed into other scenes.

Potential solutions for issues with viewpoint stability over multiple frames and performance have also been suggested. The results are encouraging, as it is shown that neural networks can learn to reproduce the blobbiness of metaballs and their blends, while also producing convincing normals.

CHAPTER 8

Summary

To summarize, this thesis dealt with the creation of a neural network that is able to produce an image depicting a scene of metaballs, producing the bonds and blends characteristic to metaballs.

First, the current state of metaball rendering and neural networks, especially in regards to image processing and image generation were discussed in Chapter 2.

Afterwards, implicit surfaces, a category to which metaballs belong, were discussed. We described how they can be defined mathematically, and the basic ways they are usually extracted and rendered. Then the history and mathematical definition of metaballs, a special case of an implicit surface, were described.

This was followed by a section on neural networks. Building up from the basic units of perceptrons, the principles of modern deep neural networks were explained. In particular, convolutional neural networks were discussed, as they build the basis for the types of neural networks that have seen great success in image related tasks over the past few years. Finally, generative adversarial networks were examined in general, as well as one particular type of GAN, the conditional GAN. These networks have shown good results in generating images from scratch.

Deferred shading was also discussed, as the goal was to have the neural network produce the G-Buffers that can be combined using a deferred shading renderer.

In Chapter 4 the conceptual framework of our system was discussed, beginning with the question of how to generate a dataset that can be used to train the network for the task of generating metaball images. Next, the planned architecture of the neural network itself was shown. We also shortly discussed how the wanted output of the network could be used in an interactive renderer.

With the conceptual framework in place, the actual implementation of the entire pipeline was discussed in Chapter 5. First, the prerequisite software and frameworks were listed.

Then, a section of how the dataset was actually generated, including the abstract input images as well as the target output images, was given. The implementation of the network itself followed, and how its loss layers were designed. Finally, it was shown how the interactive demo application using the renderer was implemented.

In the following chapter, the results of this work were presented. We showed examples of images the network produced, how the model performed, and its training progress.

Finally, the aspects of the outcome were discussed in Chapter 7. The results were found to be promising, as many aspects of the metaballs produced by the neural network fit expectations. However, there are still some issues with the results, and suggestions on how the pipeline could be improved were also discussed.

Bibliography

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[ACB17] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[ADL06] Bart Adams, Philip Dutré, and Toon Lenaert. Particle splatting: Interactive rendering of particle-based simulation data. 2006.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. CRC Press, 2008.

[BB97] Jules Bloomenthal and Chandrajit Bajaj. *Introduction to implicit surfaces*. Morgan Kaufmann, 1997.

[Bli82] James F Blinn. A generalization of algebraic surface drawing. *ACM transactions on graphics (TOG)*, 1(3):235–256, 1982.

[Bol10] Naveen Kumar Reddy Bolla. *High Quality Rendering of Large Point-based Surfaces*. PhD thesis, International Institute of Information Technology Hyderabad, 2010.

[CC18] Robert Chesney and Danielle Keats Citron. Deep fakes: a looming challenge for privacy, democracy, and national security. 2018.

[Che95] Evgeni Chernyaev. Marching cubes 33: Construction of topologically correct isosurfaces. Technical report, 1995.

[CJC+19] Yang-Jie Cao, Li-Li Jia, Yong-Xia Chen, Nan Lin, Cong Yang, Bo Zhang, Zhi Liu, Xue-Xiang Li, and Hong-Hua Dai. Recent advances of generative adversarial networks in computer vision. *IEEE Access*, 7:14985–15006, 2019.

[CS219] Standford CS231n. Convolutional Neural Networks for Visual Recognition, 2019. [Online; accessed 14-July-2019]. URL: http://cs231n.github.io/neural-networks-1/.

[CSAT17] Jinghui Chen, Saket Sathe, Charu Aggarwal, and Deepak Turaga. Outlier detection with autoencoder ensembles. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 90–98. SIAM, 2017.

[CYS15] Zezhou Cheng, Qingxiong Yang, and Bin Sheng. Deep colorization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 415–423, 2015.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[EBCV09] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. Visualizing higher-layer features of a deep network. *University of Montreal*, 1341(3):1, 2009.

[FTF+19] Ohad Fried, Ayush Tewari, Adam Finkelstein, Eli Shechtman, Adobe B Dan Goldman Kyle Genova, Zeyu Jin, Adobe Christian Theobalt, Dan B Goldman, Kyle Genova, Princeton Uni-versity, Christian Theobalt, Michael Zollhöfer, and Eli Shecht-man. Text-based Editing of Talking-head Video. page 14, 2019. URL: https://doi.org/10.1145/3306346.3323028, doi:10.1145/3306346.3323028.

[GEB16] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[GPP+10] Olivier Gourmel, Anthony Pajot, Mathias Paulin, Loïc Barthe, and Pierre Poulin. Fitted bvh for fast raytracing of metaballs. In *Computer Graphics Forum*, volume 29, pages 281–288. Wiley Online Library, 2010.

[GWK+18] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–377, 2018.

[Hor91] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[HSK89] John C Hart, Daniel J Sandin, and Louis H Kauffman. Ray tracing deterministic 3-d fractals. In *ACM SIGGRAPH computer graphics*, volume 23, pages 289–296. ACM, 1989.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[IZZE17] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *arXiv preprint*, 2017.

[JC06] Gunnar Johansson and Hamish A Carr. Accelerating marching cubes with graphics hardware. In *CASCON*, volume 6, page 39. Citeseer, 2006.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KSES12] Michael Krone, John E Stone, Thomas Ertl, and Klaus Schulten. Fast visualization of gaussian density surfaces for molecular dynamics and particle system trajectories. *EuroVis-Short Papers*, 2012:67–71, 2012.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[KSN08] Yoshihiro Kanamori, Zoltan Szego, and Tomoyuki Nishita. Gpu-based fast ray casting for a large number of metaballs. In *Computer Graphics Forum*, volume 27, pages 351–360. Wiley Online Library, 2008.

[LBB+98] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LC87] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM siggraph computer graphics*, volume 21, pages 163–169. ACM, 1987.

[LHBB99] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer, 1999.

[Lin70] Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7, 1970.

[LTH+17] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 105–114. IEEE, 2017.

[MGE07] Christoph Müller, Sebastian Grottel, and Thomas Ertl. Image-space gpu metaballs for time-dependent particle data sets. In *VMV*, pages 31–40, 2007.

[MO14] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[MP69] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 1969.

[NAM+17] Oliver Nalbach, Elena Arabadzhiyska, Dushyant Mehta, Hans-Peter Seidel, and Tobias Ritschel. Deep shading: Convolutional neural networks for screen-space shading. 36(4), 2017.

[NN94] Tomoyuki Nishita and Eihachiro Nakamae. A method for displaying metaballs by using bézier clipping. In *Computer Graphics Forum*, volume 13, pages 271–280. Wiley Online Library, 1994.

[Nyb11] Daniel Nyberg. *Analysis of Two Common Hidden Surface Removal Algorithms, Painter's Algorithm & Z-Buffering*. PhD thesis, 2011. URL: `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-130849`.

[RDS+14] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2014. `arXiv:arXiv:1409.0575`.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[SI12] László Szécsi and Dávid Illés. Real-time metaball ray casting with fragment lists. In *Eurographics (Short Papers)*, pages 93–96, 2012.

[Sil18] Thalles Silva. An intuitive introduction to Generative Adversarial Networks (GANs), 2018. [Online; accessed 14-July-2019]. URL: `https://www.freecodecamp.org/news/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394/`.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[Sob14] Irwin Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.

[SSKS17] Supasorn Suwajanakorn, Steven M Seitz, and Ira Kemelmacher-Shlizerman. Synthesizing obama: learning lip sync from audio. *ACM Transactions on Graphics (TOG)*, 36(4):95, 2017.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 197–206. ACM, 1990.

[Sur17] Ordnance Survey. A beginners guide to understanding map contour lines | OS GetOutside, 2017. [Online; accessed 14-July-2019]. URL: `https://getoutside.ordnancesurvey.co.uk/guides/understanding-map-contour-lines-for-beginners/`.

[SZS+13] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[Tom12] Lukasz Jaroslaw Tomczak. Gpu ray marching of distance fields. *Technical University of Denmark*, 8, 2012.

[Wik19] Wikipedia. Deferred shading — Wikipedia, the free encyclopedia, 2019. [Online; accessed 14-July-2019]. URL: http://en.wikipedia.org/w/index.php?title=Deferred%20shading&oldid=897972002.

[WVG92] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics (TOG)*, 11(3):201–227, 1992.

[XW16] Tian-Chen Xu and En-Hua Wu. View-space meta-ball approximation by depth-independent accumulative fields. In *SIGGRAPH ASIA 2016 Technical Briefs*, page 10. ACM, 2016.

[XWCL15] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[XXC12] Junyuan Xie, Linli Xu, and Enhong Chen. Image denoising and inpainting with deep neural networks. In *Advances in neural information processing systems*, pages 341–349, 2012.

[ZML16] Junbo Zhao, Michael Mathieu, and Yann LeCun. Energy-based generative adversarial network. *arXiv preprint arXiv:1609.03126*, 2016.

[Zuc16] Alan Zucconi. Volumetric Rendering: Raymarching, 2016. [Online; accessed 14-July-2019]. URL: https://www.alanzucconi.com/2016/07/01/raymarching/.