Université de Montréal

Towards Deep Unsupervised Inverse Graphics

par

Jérôme Parent-Lévesque

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

Décembre 2020

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Towards Deep Unsupervised Inverse Graphics

présenté par

Jérôme Parent-Lévesque

a été évalué par un jury composé des personnes suivantes :

Yoshua Bengio
(président-rapporteur)

Aaron Courville
(directeur de recherche)

Derek Nowrouzezahrai
(codirecteur)

Pierre Poulin
(membre du jury)

Résumé

Un objectif de longue date dans le domaine de la vision par ordinateur est de déduire le contenu 3D d'une scène à partir d'une seule photo, une tâche connue sous le nom d'*inverse graphics*. L'apprentissage automatique a, dans les dernières années, permis à de nombreuses approches de faire de grands progrès vers la résolution de ce problème. Cependant, la plupart de ces approches requièrent des données de supervision 3D qui sont coûteuses et parfois impossible à obtenir, ce qui limite les capacités d'apprentissage de telles œuvres. Dans ce travail, nous explorons l'architecture des méthodes d'*inverse graphics* non-supervisées et proposons deux méthodes basées sur des représentations 3D et algorithmes de rendus différentiables distincts: les *surfels* ainsi qu'une nouvelle représentation basée sur Voronoï. Dans la première méthode basée sur les surfels, nous montrons que, bien qu'efficace pour maintenir la cohérence visuelle, la production de surfels à l'aide d'une carte de profondeur apprise entraîne des ambiguïtés car la relation entre la carte de profondeur et le rendu n'est pas bijective. Dans notre deuxième méthode, nous introduisons une nouvelle représentation 3D basée sur les diagrammes de Voronoï qui modélise des objets/scènes à la fois explicitement et implicitement, combinant ainsi les avantages des deux approches. Nous montrons comment cette représentation peut être utilisée à la fois dans un contexte supervisé et non-supervisé et discutons de ses avantages par rapport aux représentations 3D traditionnelles.

Mots clés: inverse graphics, vision par ordinateur, apprentissage non-supervisé, rendu différentiable, modélisation 3D, réseaux de neuronnes génératifs, infographie, apprentissage profond, apprentissage automatique.

Abstract

A long standing goal of computer vision is to infer the underlying 3D content in a scene from a single photograph, a task known as *inverse graphics*. Machine learning has, in recent years, enabled many approaches to make great progress towards solving this problem. However, most approaches rely on 3D supervision data which is expensive and sometimes impossible to obtain and therefore limits the learning capabilities of such work. In this work, we explore the deep unsupervised inverse graphics training pipeline and propose two methods based on distinct 3D representations and associated differentiable rendering algorithms: namely surfels and a novel Voronoi-based representation. In the first method based on surfels, we show that, while effective at maintaining view-consistency, producing view-dependent surfels using a learned depth map results in ambiguities as the mapping between depth map and rendering is non-bijective. In our second method, we introduce a novel 3D representation based on Voronoi diagrams which models objects/scenes both explicitly and implicitly simultaneously, thereby combining the benefits of both. We show how this representation can be used in both a supervised and unsupervised context and discuss its advantages compared to traditional 3D representations.

Keywords: inverse graphics, computer vision, unsupervised learning, differentiable rendering, 3D modeling, generative neural networks, computer graphics, deep learning, machine learning.

Contents

List of Tables

List of Figures

List of Abbreviations

AE	Autoencoder
BRDF	Bidirectional Reflectance Distribution Function
CNN	Convolutional Neural Network
FoV	Field of View
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
IoU	Intersection-over-Union, a metric for comparing two occupancy functions
MLP	Multi-Layer Perceptron
MSE	Mean Square Error
RGB	Red-Green-Blue, a term that refers to images with three colour channels

PDF	Probability Density Function
SDF	Signed Distance Function
VAE	Variational Autoencoder

Acknowledgments

I want to thank my advisors Aaron Courville and Derek Nowrouzezahrai for their continued help and guidance. I am eternally grateful for the opportunity you both have provided me and for your never ending support in any project I undertook.

Special thanks to my colleagues, friends and co-authors (in alphabetical order): Andrea Tagliasacchi, Bhairav Mehta, Breandan Considine, Cengiz Öztireli, Christopher Beckham, Daniele Panozzo, David Vazquez, Fabrice Normandin, Fahim Mannan, Florian Golemo, Florian Shkurti, Francis Williams, Joey Litalien, Kevin Xie, Krishna Murthy, Kwang Moo Yi, Makesh Narsimhan, Martin Weiss, Miles Macklin, Sai Rajeswar, Samuel Lavoie, Peter Quinn, Sanja Fidler, Vikram Voleti, Weiwei Sun, as well as many more at the Mila and MGL labs.

Thank you to my family and close friends for their continued support and encouragement. Chantal, Jean Benoît, Laurent, Simone, Élie, Peter, Catherine, Antoine, Serena, Kevin, Alec, Jocelyn, Victor, Benjamin, Thomas, Anthony, Nicolas, Nikita; what would I do without you, especially in these tough times!

Chapter 1

Introduction

In recent years, machine learning has had a big impact on the field of computer vision. In particular, convolutional neural networks have proved to excel at image classification, segmentation and object detection tasks. Nevertheless, these methods still struggle on tasks related to 3D scene understanding. One such task that has gained a lot of attention in the last couple of years is the task of inferring a 3D scene from a number — as low as one — of 2D images. In the computer graphics field, this problem has been known and explored well before convolutional neural networks gained popularity (e.g., Shape-from-Shading as introduced by Horn in 1989 [**21**]). The arrival of machine learning and related algorithms enabled a new wave of promising research and progress on this task through a process known as *vision as inverse graphics*. In this setting, illustrated in Figure 1.1, rendering and neural networks are combined in a single system enabling end-to-end unsupervised training using natural images. The two key challenges to this method are: 1) to disentangle the learned camera parameters, lighting, materials and scene geometry such that they are physically accurate and can be individually manipulated, and 2) to ensure that every step of the process is differentiable to allow gradient back-propagation from the loss computation to all learned parameters.

In this thesis we present a review of recent work on the issue of disentanglement and discuss their shortcomings. We then propose a new method that leverages Generative Adversarial Networks and a point cloud representation to attempt to disentangle the camera parameters from the other scene parameters.

To address the second key challenge mentioned above, we also explore how the rendering process of the inferred 3D scene can be made differentiable. We discuss recent work on developing differentiable rendering algorithms and on developing 3D scene representations for which rendering is trivially differentiable. We then take a close look at two such representations: a point-cloud/surfel representation and a novel Voronoi-based representation.

Inverse graphics holds the promise for many exciting applications and a truly unsupervised approach would enable learning on a large scale required for industrial applications.

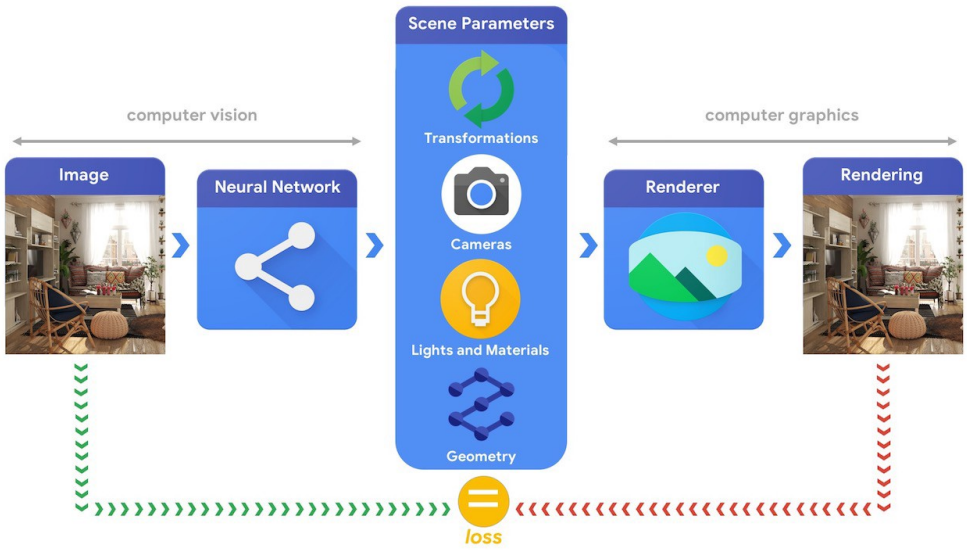

Fig. 1.1. The vision as inverse graphics pipeline. In the first half of the process, computer vision techniques such as convolutional neural networks are used to infer a scene description from a 2D image. A differentiable renderer is then used to produce a rendering of that scene. The process is self-supervised by computing a loss between the generated image to the input image. Image from TensorFlow, "Introducing TensorFlow Graphics" page (`https://blog.tensorflow.org/2019/05/introducing-tensorflow-graphics_9.html`)

Some key examples of such applications include video editing and visual effects, image manipulation, interactive 3D photo visualization, human pose estimation (Figure 1.2), better vision and mapping systems for robotics and autonomous driving, photogrammetry, improved realism in video games and many others. Additionally, there is evidence for vision as inverse graphics and *analysis-by-synthesis* to be a key part of the functioning of the brain's vision system [**71**]. We hope that this work can provide some insights into potential paths towards this compelling goal by not only exploring new ideas, but also by thoroughly analyzing both their benefits and shortcomings.

This thesis is structured as follows: After presenting a review of relevant computer graphics and machine learning concepts in Chapter 2, we introduce in Chapter 3 a method that attempts to address the issue of disentangling camera position from 3D scene while allowing novel-view synthesis. In Chapter 4, we present a new differentiable 3D representation based on Voronoi diagrams and explore its benefits compared to other popular representations. We conclude by restating our key contributions and discussing avenues for future work in Chapter 5.

Fig. 1.2. Example of an application of inverse graphics: Human pose estimation from a single color image trained using a differentiable 3D mesh renderer by Pavlakos *et al.* [**50**]

Chapter 2

Background

In this chapter we present an overview of basic concepts in the fields of computer graphics and machine learning related to our work on inverse graphics. We introduce a mathematical foundation (and notation) that we will then build upon in later chapters. We begin by summarizing core ideas in the field of 3D rendering, followed by a description and comparison of common 3D representations with a particular focus on their applicability to deep learning systems. We then finish with a summary of generative models for unsupervised learning, including autoencoders and generative adversarial networks (GANs).

2.1. 3D Rendering

The field of 3D Rendering is vast, constantly evolving and spans many areas of application with different constraints. In this section we cover only the basics of physically-based rendering, differentiable rendering and volume rendering. Whenever possible, we provide references to relevant books and surveys that we invite the reader to consult for further details.

In rendering, the goal is to produce an accurate 2D image of a 3D scene as seen by a virtual camera. It can be desired to achieve a photo-realistic image, but artistic visualizations or stylized images are also often required for applications such as video games and 3D animation movies. At its core, rendering is a light transport simulation that computes how light bounces in a scene starting from emitters and eventually reaching the viewer's eyes or camera's sensor. Although some specific applications deviate slightly from this definition, most rendering techniques achieve their goal through some approximation of this light transport simulation. Indeed, different applications put different constraints on the rendering algorithm which in turn puts constraints on the quality of the simulation approximation. A video game, for example, will usually require an image to be rendered at an interactive frame-rate such as 60 frames-per-second whereas a movie production may have access to a much larger time and compute budget allowing for more complex simulations.

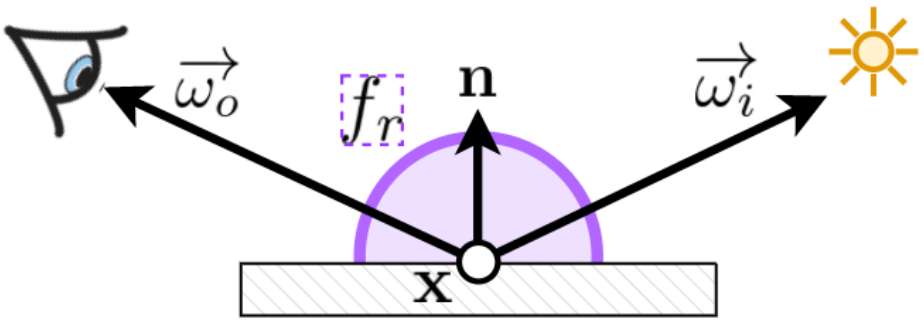

Fig. 2.1. Illustration of the different quantities involved in the rendering equation. The outgoing radiance L_o is computed along the vector $\overrightarrow{\omega_o}$ and the incoming radiance L_i along the vector $\overrightarrow{\omega_i}$. The BRDF function f_r is defined on the hemisphere (in purple) around normal $\mathbf{n}$ at point $\mathbf{x}$. While in this simple example, the light path from the emitter to the viewer/camera consists of a single bounce, it is usually the case that light paths consist of many more bounces.

2.1.1. Physically-based Rendering

In his seminal 1986 paper [**25**], Kajiya introduced the rendering equation to generalize a variety of rendering algorithms and to provide a mathematical foundation for the simulation of light transport. The rendering equation proposed in that paper is as follows:

$$L_o(\mathbf{x}, \overrightarrow{\omega_o}) \;=\; L_e(\mathbf{x}, \overrightarrow{\omega_o}) \;+\; \int_\Omega f_r(\mathbf{x}, \overrightarrow{\omega_i}, \overrightarrow{\omega_o})\, L_i(\mathbf{x}, \overrightarrow{\omega_i})\, (\overrightarrow{\omega_i} \cdot \mathbf{n})\, \mathrm{d}\, \overrightarrow{\omega_i} \tag{2.1.1}$$

Note that we ignore here and throughout this thesis the effect of varying light wavelengths for clarity and simplicity. Computations are usually carried out at a specific wavelength.

This equation relates the outgoing radiance L_o at a surface point $\mathbf{x}$ in the direction $\overrightarrow{\omega_o}$ to the emitted radiance L_e at that point and to the integral of the incoming radiance L_i over a hemisphere Ω around the surface normal $\mathbf{n}$. $\mathbf{n}$, $\overrightarrow{\omega_o}$ and $\overrightarrow{\omega_i}$ are unit vectors representing directions. The output L_o is a function of the surface point $\mathbf{x}$ and view direction $\overrightarrow{\omega_o}$. In practice, the surface point $\mathbf{x}$ is found by taking the first intersection with the scene of a ray that starts from the viewer and that goes in the direction $-\overrightarrow{\omega_o}$. Figure 2.1 illustrates these quantities for a simple light path that connects an emitter with the viewer through a single bounce on a flat surface. The integrand part of this equation represents the radiosity, or in other words the flux (power) leaving the surface per unit area for a given direction $\overrightarrow{\omega_i}$. It consists of three terms that are multiplied together: the Bidirectional Reflectance Distribution Function (BRDF), the incident radiance and the cosine foreshortening term.

The BRDF f_r is a function that describes how light gets reflected off a surface. It varies based on the material properties and albedo at the intersection point on an object. The BRDF can be seen as describing the ratio of how much energy is absorbed versus reflected by the surface for light coming in from direction $\overrightarrow{\omega_i}$ and leaving in direction $\overrightarrow{\omega_o}$. Some

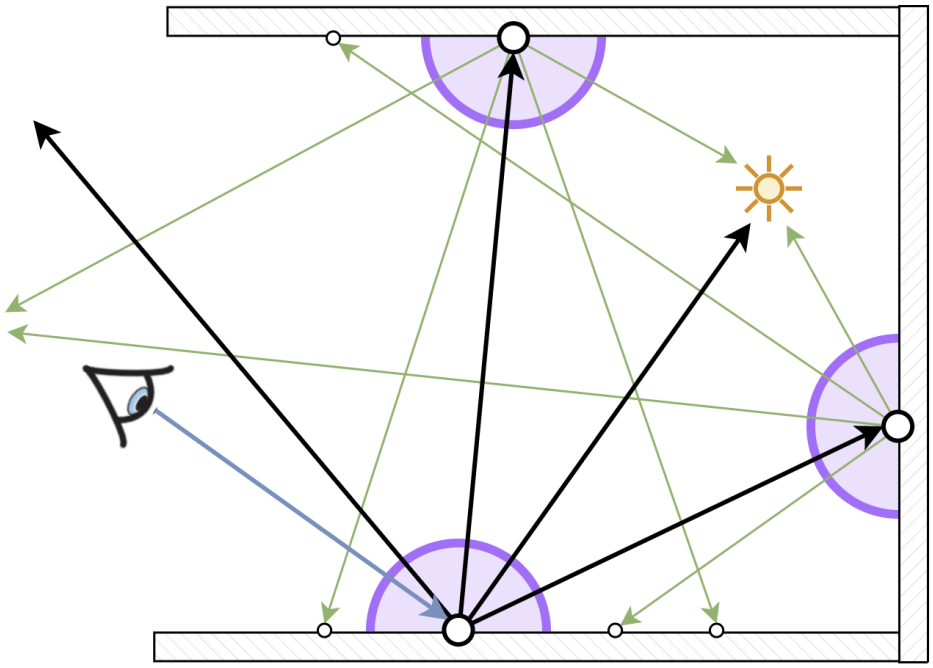

Fig. 2.2. Illustration of the path tracing process. Paths from the viewer to the emitters are traced by recursively finding intersection points of rays with the scene and tracing more rays from those points. In this figure, black arrows represent rays traced on the first bounce while green arrows represent rays traced on the second bounce. Usually the number of bounces is not limited to two and path termination is instead determined through Russian Roulette.

examples of simple common BRDF functions include the Lambertian BRDF (where ρ is the albedo)

$$f_r = \frac{\rho}{\pi} \tag{2.1.2}$$

and the Phong BRDF (where α is a scalar that controls the specularity of the material and $\cdot$ is the vector dot product)

$$f_r = \frac{\rho(\alpha+2)}{2\pi}(\overrightarrow{\omega_i} \cdot (2(\overrightarrow{\omega_i} \cdot \mathbf{n})\mathbf{n} - \overrightarrow{\omega_i}))^{\alpha} \tag{2.1.3}$$

The incident radiance L_i, as described above, represents the light coming from the scene to point $\mathbf{x}$ from direction $\overrightarrow{\omega_i}$.

Finally, the cosine foreshortening term $(\overrightarrow{\omega_i} \cdot \mathbf{n})$ models Lambert's cosine law which states that the observed radiosity is proportional to the cosine of the angle between the normal $\mathbf{n}$ and incident light direction $\overrightarrow{\omega_i}$. Intuitively, this can be seen as the ratio of the projected area of a flat surface seen at an angle by the eye over the actual area of that surface.

If a surface is not emissive then L_e is simply 0.

Assuming the scene is in a vacuum (i.e., there are no participating media that could absorb or scatter light outside of object's surfaces), then this equation can be seen as recursive. Indeed, the incoming radiance L_i at a point $\mathbf{x}$ from direction $-\overrightarrow{\omega_i}$ is itself the outgoing radiance from a different surface point $\mathbf{x}'$ that intersects the ray coming out of $\mathbf{x}$ in direction $\overrightarrow{\omega_i}$. We can therefore use this equation to model light propagation in a scene through any number of bounces off surfaces which thus allows us to model how light coming from emitters in the scene can reach the viewer or virtual camera. This process is shown in Figure 2.2.

To render an image from a given point of view, we first need to determine a model for the virtual camera. In this thesis, we use a simple model where a camera's position is the focal point. A camera is then fully defined by its location in space, its vertical or horizontal field of view (FoV), pixel resolution and a unit vector $\overrightarrow{at}$ indicating the direction the camera is pointing towards. We always assume that the camera is oriented parallel to the ground. The camera's aspect ratio can be determined from the pixel resolution. This differs from a physical pinhole camera where the focal point would be located at a different location than the camera's sensor.

The camera's resolution determines how an image signal is discretized to be stored in a regular 2D grid of pixels. In a physical camera, each pixel is captured by a light-sensitive photosite. The same discretization effect is achieved in 3D rendering by integrating the incoming radiance from the scene over the solid angle subtended by a pixel.

To render a full image, each pixel can therefore be rendered individually (potentially in parallel) by estimating the integral over the pixel's solid angle using a limited number of rays and by estimating Equation 2.1.1 at the intersection of each ray with the scene. In practice, different approximations are made depending on the application. We present here two of the most common rendering techniques.

In the rendering technique known as **Rasterization**, very rough approximations are made in order to achieve rendering speeds fast enough for real-time scenarios. The integral over a pixel's solid angle is estimated using a single ray through the center of the subtended solid angle. In addition, the integral of incident radiance L_i is usually estimated by considering only directions $\overrightarrow{\omega_i}$ that point towards emitters in the scene. No recursion is performed and the number of bounces of each ray is therefore limited to 1. To further speed up the rendering process, the scene's geometry is projected onto the camera's image plane which avoids the expensive ray-scene intersection operation that would otherwise be performed for each pixel of the image. Figure 2.3 illustrates the rasterization procedure. This process is usually performed on a graphics processing unit (GPU), a device that supports hardware-accelerated rasterization.

Path Tracing, on the other hand, is a technique that sacrifices speed for physical accuracy. Given enough time, a path tracer will converge to the exact solution to the rendering equation. This is achieved through *Monte Carlo integration* with importance sampling, a

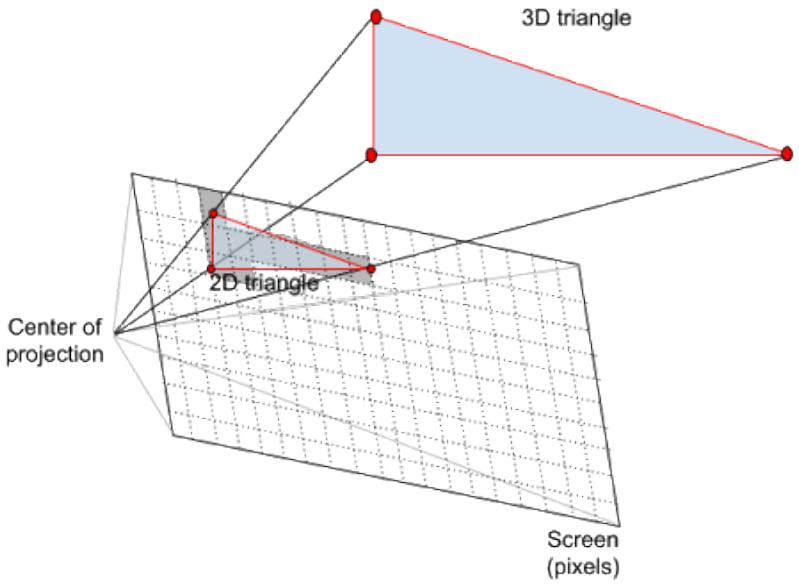

Fig. 2.3. The rasterization process. Each triangle primitive in the scene is projected to the camera plane before being rasterized to the regular grid of pixels. Image from `https://mtrebi.github.io/2017/02/01/rasterization-i.html`

numerical integration method which is used here to approximate both the incident radiance integral and the pixel solid angle integral. This method states that an integral can be approximated in the following way:

$$\int_0^1 f(x)\,dx \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} \qquad x_i \sim p(x) \tag{2.1.4}$$

using N random x_i sampled from the probability distribution function (PDF) $p(x)$. Note that in the limit where $N \to \infty$ the right hand side of this equation converges to the exact solution. In path tracing, Equation 2.1.1 is approximated using Monte Carlo integration for each ray intersection with the scene's geometry. This means that light paths consisting of multiple bounces are supported, making path tracing a *global illumination* system. Nevertheless, this leads to an infinitely recursive process; a problem that is solved in practice using a method known as *Russian Roulette*. In Russian Roulette, each ray has a probability $r \in [0, 1]$ of being terminated after each bounce. The incident radiance computed for each ray that has not been terminated is then multiplied by $\frac{1}{1-r}$. Compared to simply setting a hard limit on the number of bounces of each ray, this method does not add any bias to the rendered image.

For more information on physically-based rendering, we refer the reader to the *PBR* book by Pharr et al. [**52**].

2.1.2. Differentiable Rendering

While renderers solve the *forward* process of image synthesis, it is often desirable to go in the reverse direction. A popular approach to solve this backward process is to define differentiable rendering algorithms so that gradient-based optimization techniques can be applied. Without any changes, both rasterization and path tracing renderers are naturally differentiable with respect to material properties assuming a differentiable BRDF. Using an automatic differentiation library like PyTorch [**49**], it is therefore straightforward to modify an existing renderer implementation such that gradients can be computed with respect to the *appearance* of objects in the scene. Nevertheless, in some rendering algorithms, this can quickly become too computationally and memory intensive as the computation graphs that are generated by automatic differentiation libraries can become very large. Furthermore, more work is needed to address the problem of differentiability with respect to scene geometry, also known as *differentiable visibility*. Solving this problem is also required to be able to compute gradients with respect to camera and light parameters.

There are two sources of discontinuities in the rendering process that prevent computing visibility gradients. The first is that ray-scene intersections are binary in nature, i.e., object edges are discrete and can thus be seen as a step function (over which gradients are 0). This means that the colour of each pixel in the rendered image is not a function of the position of object vertices. The second source of discontinuity comes from the fact that surfaces being rendered are usually fully opaque. Gradients with respect to objects occluded by other objects from the point of view of the camera can therefore not be computed.

OpenDR [**36**] is the first work to show a general differentiable rasterizer that can differentiate with respect to both visibility and material properties. They achieve this by approximating the gradients with respect to vertex positions using a finite difference approach. This can be done by applying a Sobel filter on the rendered image while taking into account extra considerations for pixels that lie on object boundaries.

Neural 3D Mesh Renderer (NMR) [**29**] proposes a different approximation that is better suited for use in conjunction with neural networks. They use linear interpolation in the backward pass only, as shown in Figure 2.4. This results in non-zero gradients to the object geometry not just from pixels adjacent its boundary but from pixels that are in the general proximity. The authors show that this is beneficial to neural network training dynamics as it makes the loss landscape smoother.

Soft Rasterizer (SoftRas) [**34**] [**35**] takes this idea a step further by entirely replacing the step function shown in Figure 2.4 (b) by a smooth approximation: the sigmoid function. They achieve this by computing the signed distance between each pixel's position and the nearest mesh edge in the image plane. This distance is then passed through a sigmoid function before using the result as an opacity weight. For pixels inside objects, the distance is

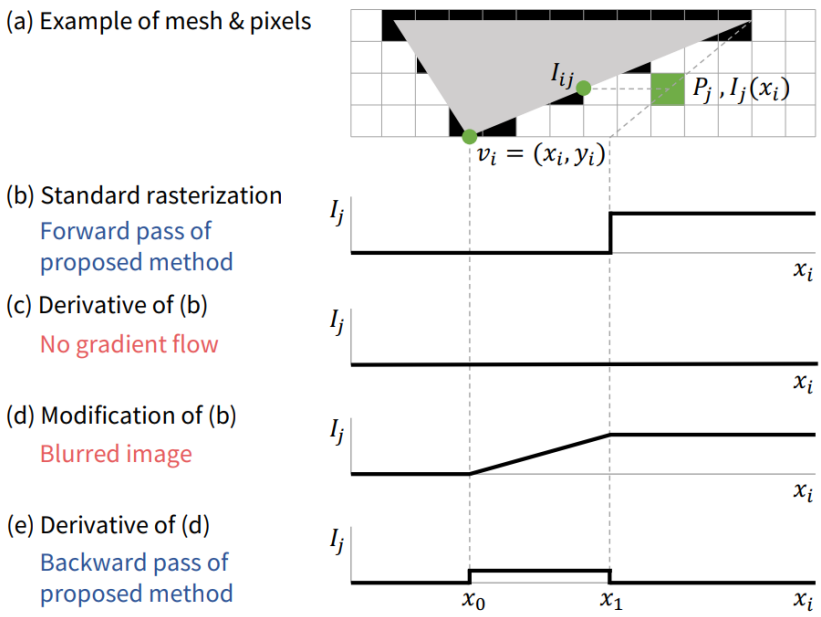

Fig. 2.4. Approximation of the gradients in Neural 3D Mesh Renderer. The forward pass (a) is the same as a standard rasterizer while in the backward pass, gradients (e) are computed from a blurred image (d). Image from NMR [**29**].

positive meaning that surface point is more opaque, while for pixels outside object boundaries the distance is negative and the surface point is more transparent. For pixels directly on the boundary, the opacity weight is 0.5. The opacity weights at a given pixel are passed through a softmax function to determine how much the radiance coming from each mesh polygon contributes to that pixel's final colour. This is the first method that addresses both sources of visibility discontinuities since occluded objects can now also get a gradient signal. While this process is slower than a traditional renderer since each polygon in the mesh contributes to the colour of all pixels in the image, it provides non-zero gradients to the object geometry at every pixel.

More recently, Differentiable Interpolation-based Renderer (DIB-R) [**8**] proposes a variant of SoftRas that treats pixels on the inside of objects (foreground pixels) differently than pixels outside of the objects' boundary (background pixels) for rendering objects with vertex-defined colours. Gradients for background pixels are computed using the same trick as SoftRas except that the opacity weights are now computed by taking the exponential of the negative distance to the nearest edge. This opacity weight is thus one on object boundaries and decays to zero when moving away from the object. In DIB-R, foreground pixels are naturally differentiable since a pixel's colour is computed through barycentric interpolation of the face's vertex colours (a differentiable process itself).

Similarly, numerous differentiable path-tracing algorithms that support differentiable visibility have been proposed in recent literature. Li et al. (Redner) [**33**] and Zhang et al. [**73**] propose to split the incident radiance into an interior integral and a boundary integral where the latter is explicitly sampled. In an approach integrated in the popular Mitsuba2 rendering software [**46**], Loubet et al. [**38**] propose to reparameterize the integrand such that Monte Carlo sampling is differentiable by performing it relative to an object's boundary. These approaches differ from some of the differentiable rasterizers in that all of them attempt to perfectly match the ground-truth derivatives without considering the effect of a non-smooth loss landscape on the optimization process and on neural network training.

Finally, some work has also been done to accelerate the gradient computation pass of differentiable renderers. As discussed earlier, the efficiency of differentiable rendering is memory-bound as the computation graphs can quickly get very large. This is especially true of differentiable path-tracing algorithms. In *Radiative backpropagation* [**45**], Nimier-David et al. propose to drastically lower the memory requirements of such algorithms by treating the backward pass as a transport simulation of partial derivatives. This pass can be computed in the same way as the forward rendering process, with the only difference that derivatives are propagated instead of radiance. By removing the memory bottleneck, the authors show that the rendering speed can be increased by almost two orders of magnitude on current hardware. Nevertheless, this method does not support differentiable visibility and thus can only be used to optimize material properties.

For more information on differentiable rendering, we refer the reader to Kato et al.'s differentiable rendering survey [**28**] and to the 2020 SIGGRAPH course on the subject [**74**].

In this section we presented an overview of differentiable rendering techniques applicable to *mesh-based 3D representations*. This is by far the most prominent type of representation used in the field of computer graphics. Nevertheless, as we will see in later sections, other types of representations may be better suited to inverse graphics applications. Such representations present different differentiable rendering challenges.

2.1.3. Volume Rendering

Up until this point we have only discussed rendering of scenes that are in a vacuum and where objects are fully solid and opaque. When either of these two assumptions break, for example to model smoke or atmospheric effects, one needs to use volume rendering techniques.

When there is a *participating medium* in the scene, radiance changes along a ray as opposed to changing only at intersecting points. There are four mechanisms by which this change occurs. First, *absorption* is the process by which the participating medium absorbs some of the radiance along the ray. Second is *out-scattering*, i.e., radiance along the ray that

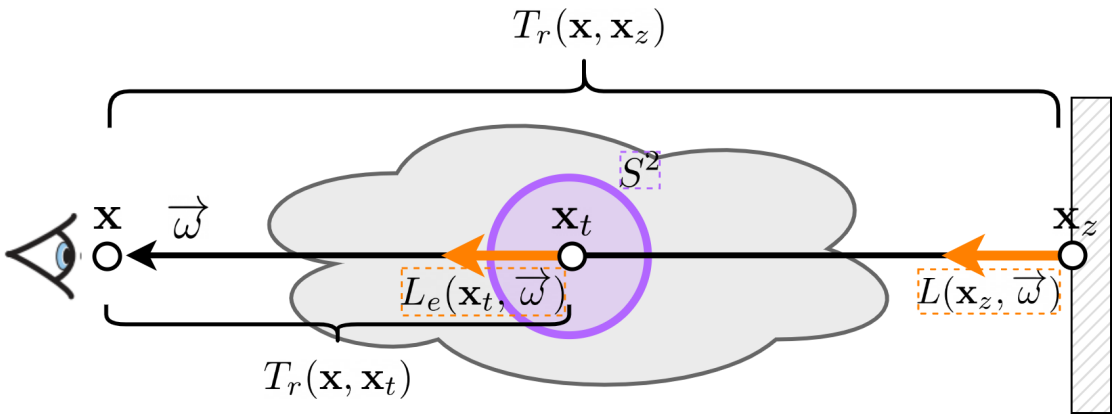

Fig. 2.5. Illustration of the various quantities involved in volume rendering (Equation 2.1.5). Pictured here is a single light ray coming from a wall (on the right) towards the viewer/virtual camera (on the left). The light goes through a participating medium shown with a cloud icon. For all points along the ray within the cloud, one must compute an integral of the incident radiance over the sphere S^2 shown here in purple.

gets scattered in a different direction. Similarly, *in-scattering* is the third process where this time radiance coming from directions other than the ray at hand gets scattered in the same direction as that ray. Finally, the fourth and last mechanism by which radiance can change along a ray is *emission* where the participating medium produces light itself.

Combining these four systems together we get the following volume rendering equation:

$$\begin{aligned} L(\mathbf{x}, \overrightarrow{\omega}) &= T_r(\mathbf{x}, \mathbf{x}_z) L(\mathbf{x}_z, \overrightarrow{\omega}) \\ &+ \int_0^z T_r(\mathbf{x}, \mathbf{x}_t) \sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \overrightarrow{\omega}) dt \\ &+ \int_0^z T_r(\mathbf{x}, \mathbf{x}_t) \sigma_s(\mathbf{x}_t) \int_{S^2} f_p(\mathbf{x}_t, \overrightarrow{\omega}', \overrightarrow{\omega}) L_i(\mathbf{x}_t, \overrightarrow{\omega}') d\overrightarrow{\omega}' dt \end{aligned} \tag{2.1.5}$$

where $\mathbf{x}$ and $\overrightarrow{\omega}$ define the ray, z is the length of that ray, t is a distance along the ray, L is the output radiance, L_e is the emitted radiance, L_i is the incident radiance, σ_a is the absorption coefficient, σ_s is the scattering coefficient, S^2 is the unit sphere and f_p is the *phase function*. The phase function is the volume equivalent of the BRDF on surfaces as it defines the scattering behaviour of light in the medium. $T_r(\mathbf{x}_i, \mathbf{x}_f)$ is the transmission, or in other words the ratio of how much light makes it through the path $\mathbf{x}_i \to \mathbf{x}_f$. It is defined by the following equation:

$$T_r(\mathbf{x}_i, \mathbf{x}_f) = e^{-\int_0^{||\mathbf{x}_i - \mathbf{x}_f||} \sigma_t(t)\, dt} \tag{2.1.6}$$

where σ_t is the *extinction coefficient* and is the sum of the absorption coefficient σ_a and the (out-)scattering coefficient σ_s. It therefore represents the total loss of radiance per unit distance). In a homogeneous medium, σ_t is constant and Equation 2.1.6 can be simplified to:

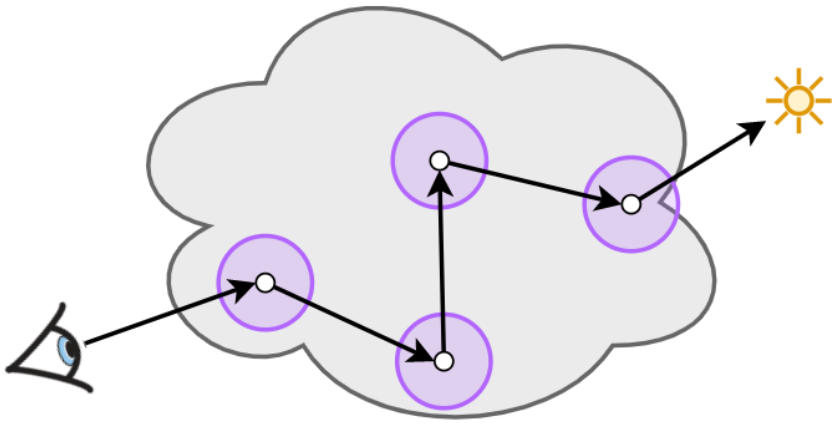

Fig. 2.6. Illustration of the volumetric path tracing process. To estimate scattering, paths are chosen in the volume by randomly sampling a direction and distance at each scattering step.

$$T_r(\mathbf{x}_i, \mathbf{x}_f) = e^{-\sigma_t ||\mathbf{x}_i - \mathbf{x}_f||} \tag{2.1.7}$$

The first term in Equation 2.1.5 corresponds to the absorption and out-scattering mechanisms (the losses of light), the second to the emission of the participating medium and the third to the in-scattered radiance. The various quantities in this equation are illustrated in Figure 2.5.

In practice and similarly to surface rendering techniques presented in Section 2.1.1, this equation is also approximated in various ways. For example, the scattering term can be entirely omitted or an unbiased estimate can be obtained, although at significant computational cost, through Monte Carlo integration. This volumetric path tracing process is also recursive, similarly to the traditional surface-only path tracing method described in Section 2.1.1. Figure 2.6 illustrates the global view of volume rendering and how it can be approximated using path tracing.

For more information on volume rendering, we refer the reader to the volume scattering chapter of the PBR book [**52**].

In a scene made entirely of various participating media, the two sources of discontinuity explored in section 2.1.2 that caused issues with differentiable rendering do not exist as there are no sharp object edges. In this case, the rendering process is naturally differentiable.

2.2. 3D Representations for Deep Learning

As explained in the previous section, fully volumetric renderings are naturally differentiable. For this reason, volume representations of scenes have been explored as alternatives to

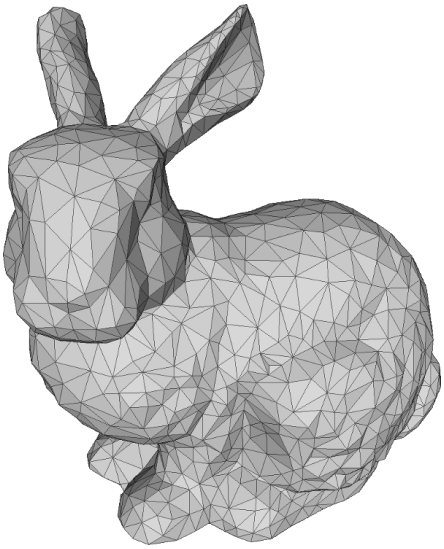

Fig. 2.7. Example of a bunny model represented using a triangle mesh.

mesh-based representations in deep learning applications. This raises the question: What is the best 3D representation for inverse graphics tasks? In this section, we present an overview of the most popular 3D representations in the recent literature and weigh their pros and cons in the context of unsupervised 3D deep learning.

2.2.1. Meshes

Polygon meshes are by far the most common 3D representation in the field of computer graphics. They are the default in the video game and computer animation industries, and are the primary support focus of most rendering, animation and 3D modeling software. More specifically, triangle meshes are the most widespread type of polygon mesh and most hardware-accelerated rasterizers primarily support triangle primitives. Meshes are a surface representation. They do not model volumetric quantities although the surface is typically closed and thus contains a volume. An example of a model represented using a triangle mesh is shown in Figure 2.7.

Meshes consist of three components: vertices, edges and faces. For example in a triangle mesh, the triangles form the faces and share edges and vertices with neighbouring triangles. These components are mathematically connected through Euler's formula (for closed surfaces):

$$V + F - E = 2 - 2g \tag{2.2.1}$$

where V is the number of vertices on the mesh, F is the number of faces and E is the number of edges. g is the *genus* of the surface. In simple terms, this is an integer representing the number of "holes/handles" the surface contains. For example, a sphere has genus 0 while a

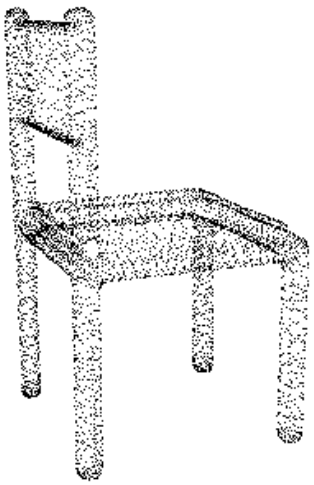

Fig. 2.8. Example of a chair model represented using a point cloud.

coffee mug has genus 1. All surfaces that share the same genus number can be reshaped into each other. Following the same example, a coffee mug can be smoothly morphed into a torus shape (which also has genus 1) without changing the *connectivity/topology* of the mesh. This relationship is known as *homeomorphy*. Similarly, the genus of a surface cannot be changed without changing the connectivity of the mesh. This follows from Equation 2.2.1.

Mathematically, the vertices and edges in a mesh form an embedded undirected graph. The edges in the graph determine the connectivity of the mesh. This is unaffected by changes in any or all of the vertices' position in space.

Recent work such as SoftRas [**34**] [**35**] and DIB-R [**8**] have successfully used triangle mesh representations in combination with differentiable rasterizers to infer a 3D model of an object from a single color or silhouette image. They achieve this by using neural networks that output vertex positions for a predefined mesh. Nevertheless, as discussed above, since the connectivity of such a mesh does not change these models are limited to model objects that have a fixed genus — usually 0. The main advantages of polygon mesh representations is their very widespread support and number of pre-existing software and algorithms for downstream tasks (including rasterization for real-time rendering). Furthermore, compared to other types of representations, meshes explicitly model surfaces and thus directly provide an object's boundary.

2.2.2. Point Clouds

A different type of 3D representation is point clouds, a sparse representation where scene information is stored in an unordered set of points. Typically, this is used as a surface representation like polygon meshes by storing only surface points in the set. Nevertheless,

point clouds can also be used to represent volumes. This comes at a very high memory cost because of the curse of dimensionality — a much larger number of points is required to achieve similar density in 3D as the surface (2D) representation. For this reason, we only explore in this section point clouds used as to model surfaces. An example of a model represented using a point cloud is shown in Figure 2.8.

With each point can be stored some additional data, typically in the form of a fixed-length vector. This allows easy storage of the whole point cloud in a single matrix, and makes it easily compatible with tensor computation libraries such as PyTorch [**49**]. For example, points can store color or a set of material properties. In some machine learning applications, a learned latent vector is associated to each point such as in PointNet [**53**] and PointNet++ [**55**]. Both of these methods employ a special neural network architecture to explicitly handle the unordered nature of points in the point clouds they use as input. Their core idea is to process each point in parallel using a shared neural network and to then use a permutation-invariant operator such as max() over the learned latent vectors produced as outputs for each point. Other methods propose to attach latent descriptors to points in an effort to learn a machine learning model that can render photo-realistic images from these augmented point clouds [**1**].

Another useful property to store for each point is the normal vector of the surface at that point. Having this value makes it much easier to then reconstruct the surface from the sparse set of points with techniques such as Poisson Surface Reconstruction [**30**]. When surface normal information is stored in point clouds, the points in the representation may be named *surfels* [**51**].

Point clouds are a flexible representation as they are not limited by topology and genus like meshes. They can represent any surface and are also easy to output from a neural network since they can be stored as a fixed-size matrix. Nevertheless, this assumes that the number of points in the set is predetermined and kept constant. In generative machine learning models using this form of output, this means that all generated objects will be represented using the same number of points. This method thus does not take into account that objects with more complex geometry require a more dense sampling of points for an accurate representation. Some methods get around this issue by modelling the surface as a distribution from which points can be sampled. This is the case in PointFlow [**69**] where a normalizing flow model is employed to learn a generative model of point clouds.

Compared to other types of representation, point clouds often require a lot of memory for accurate representations. Additionally, it is expensive and not trivial to reconstruct surfaces from a point cloud, particularly if the points are very sparse. They can, however, represent surfaces of any topology and are neural-network friendly.

Rendering point clouds can easily be done in a similar fashion as rasterization of polygon meshes. Each point in the set can individually be projected (*splatted*) to the camera plane

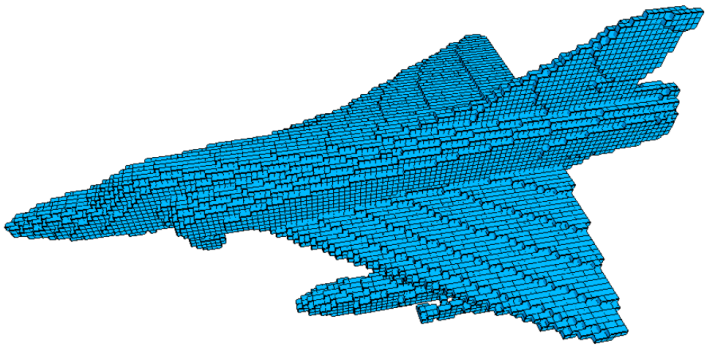

Fig. 2.9. Example of an airplane model represented using a voxel grid.

to form the final image. Nevertheless, a size for each splat on the camera plane needs to be determined. This can be a fixed constant or can be computed in order to ensure that there are no holes in the final image, especially in more sparse regions. To make this rendering process differentiable and therefore to allow the use of point cloud representations in inverse graphics applications, several approaches have been explored. Insafutdinov et al. [**22**] propose to model splats as gaussians instead of circles of fixed size that lead to discontinuities. In Differentiable Surface Splatting, Yifan et al. [**70**] propose an alternative approach based on screen space elliptical weighted average filtering.

Interestingly, an RGB-D image (an RGB image with an associated depth map) can be seen as a point cloud. The depth value for each pixel can be used to back-project each pixel along their corresponding camera ray to a separate point in 3D space. This is an idea explored in Pix2Shape [**56**] where a generative model conditioned on the camera position outputs a fixed number of points corresponding to the number of pixels in the target image. This allows outputting a point cloud that is always the same size while making sure points are visible from the camera point of view. Nevertheless, the fact that only part of the scene geometry is modeled makes it hard to extract complete surfaces and to render while taking global illumination into account.

For more information about point cloud representations in the field of computer graphics we refer the reader to the survey by Kobbelt et al. [**32**].

2.2.3. Voxels

Another type of representation, and one of the first representations that were applied to 3D deep learning, is voxels. The term *voxel* refers to individual elements in a regular grid that divides a space into box cells of identical size. The whole representation is thus often referred to as a *voxel grid*. Each voxel element can store scalar data such as an occupancy bit or vector values such as colors or abstract representations. Usually each element's data

has the same size as this allows for a particularly efficient storage management system (the whole grid can be stored in a single dense Tensor). In 3D ShapeNets [**68**] for example, a voxel grid is used to store a probability distribution of occupancy values. An example of a model represented using a voxel grid is shown in Figure 2.9.

Voxel grids' main advantage over other representations is their simplicity. Their fixed size and regular grid pattern makes them particularly well suited for deep learning applications. In particular, voxels are the natural extension to 3D space of 2D images and as such can be easily used in a convolutional architecture, in this case a 3D CNN. This is exactly what was done in VoxNet [**39**] in order to perform 3D object recognition.

The biggest drawback of this representation is the high memory and compute requirements. This is exemplified in VoxNet where the authors highlight that the maximum voxel grid resolution that they were able to process was 32^3, and in 3D ShapeNets were the resolution used was 30^3. Although it is now possible to run larger models on newer hardware, at such a resolution only a very coarse shape can be represented by this grid.

Qi et al. [**54**] investigate the performance gap between 3D CNNs and multi-view CNNs and propose multiple improvements to both approaches to bring them closer while improving upon the state-of-the-art in object classification. In particular, they find that adding an auxiliary task where the 3D CNN model has to perform classification from a subvolume in the voxel grid provides a significant performance benefit.

Wang et al. [**66**] propose a more memory efficient approach by leveraging an Octree data structure, drawing inspiration from the field of computer graphics. They further design a convolutional architecture that is able to handle this type of representation.

A final benefit of voxels is that they natively represent volumetric quantities, which makes them particularly well-suited for use in the inverse graphics pipeline alongside a (naturally differentiable) volume renderer. Additionally, it is relatively easy to extract a surface out of an occupancy voxel grid although still at a significant computational cost, especially at high grid resolutions.

2.2.4. Implicit Representations

Finally, the last category of representations that we will present here is implicit representations. This type of representation has gained a lot in popularity in the past couple of years because of the visually impressive results it produced and because of its numerous benefits. These include compactness in terms of memory, the fact that all topologies can be represented and that it can efficiently model both volumetric quantities and surfaces.

More specifically, we are interested here in neural implicit representations, where the representation is stored as the weights of a neural network. This neural network is typically

a Multi-Layer Perceptron (MLP) with learned parameters θ and its storage requirement is usually under 10MB.

Implicit representations can model surfaces as *implicit surfaces*. These are a well defined and explored concept in both the fields of mathematics and computer graphics. A surface is defined as the zero level-set of a function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ that maps a coordinate in space $\mathbf{x}$ to a real value and is thus defined by the following equation:

$$f(\mathbf{x}) = 0 \tag{2.2.2}$$

While the output value of this function outside of the zero level-set can be arbitrary, in computer graphics this is usually interpreted as a signed distance, making f a *signed distance function* (SDF). In this type of function, the output value at a point in space represents the euclidean distance to the nearest surface. Assuming that the surface is closed, the value is also given a positive sign if the point lies inside the surface, and a negative sign if the point lies outside of it. This formulation is used in the work by Park et al. (DeepSDF) [**48**] where a neural network is trained using direct 3D supervision to represent the signed distance function of a given scene. Additionally, DeepSDF explores the use of an auto-encoder architecture to create a generative shape model. In this model, the decoder neural network f has learned parameters θ and the coordinate input $\mathbf{x}$ is augmented with a latent code $l \in \mathcal{L}$:

$$f_\theta : \mathbb{R}^3 \times \mathcal{L} \rightarrow \mathbb{R} \tag{2.2.3}$$

In Sign Agnostic Learning (SAL), Atzmon et al. [**3**] [**4**] propose an alternative to the problem that approaches such as DeepSDF requires ground truth signed distance functions which are hard and expensive to compute. Their approach is able to accurately learn a signed distance function only from a sparse surface point cloud input, with or without surface normal information. Finally, Davies et al. [**10**] explore in more detail how a neural network can be trained to overfit the SDF of a single object or scene.

Instead of a signed distance function, one can choose to model implicit surfaces using an *occupancy function* $f : \mathbb{R}^3 \rightarrow \{-1,1\}$, where the output is discrete and is either positive or negative 1. A positive value means that the point lies inside of the surface. This function definition can be slightly altered to allow for a continuous output in $[-1,1]$ which makes it better suited for use in neural networks. This is the formulation used in the Occupancy Network (OccNet) paper by Mescheder et al. [**41**]. In OccNet, the neural network is trained using 3D supervision in the form of a point cloud with ground truth occupancy values. Using a dataset of 3D meshes from which they are able to extract occupancy values in random points in space, OccNet is able to learn a variational autoencoder model for basic shape categories like chairs. The use of 3D supervision made it possible to train this model without having to perform any differentiable 3D rendering.

Fig. 2.10. Example photo-realistic images rendered from viewpoints unseen during training using the NeRF neural implicit representation. Image taken from NeRF [**42**].

Neural implicit representations can have more than one output and as such can be used to additionally model various volumetric quantities. In Scene Representation Networks (SRN) [**63**], the neural network outputs a latent vector that is used by a learned linear layer to produce a final rendered colour for a given pixel in the target image. A similar final rendering layer is applied in Pixel-Aligned Implicit Functions (PIFu) [**58**] but using the latent vector is generated individually for each pixel in the input image of an autoencoder configuration. In Neural Radiance Fields (NeRF) [**42**], the neural network f outputs the emitted radiance L_e in addition to the absorption coefficient σ_a at coordinate $\mathbf{x}$. This makes it possible to easily produce a differentiable rendering of the scene by solving the following equation using numerical integration (ray marching):

$$L(\mathbf{x}, \overrightarrow{\omega}) = \int_0^z T_r(\mathbf{x}, \mathbf{x}_t)\sigma_a(\mathbf{x}_t)L_e(\mathbf{x}_t, \overrightarrow{\omega})dt \tag{2.2.4}$$

which corresponds to the second term in the volume rendering equation (2.1.5). Using this technique, the authors of NeRF show that it is possible to learn a 3D representation of a full scene using only a collection of RGB images as supervision and that it is possible to produce photo-realistic renderings from this representation for novel viewpoints. See example images rendered using NeRF in Figure 2.10. In Generative Radiance Fields (GRAF) [**60**], the idea proposed by NeRF is further enhanced by incorporating it in a generative model, thus making it possible to generate novel scenes and to store multiple scenes in the parameters of a single neural network. In X-Fields by Bemana et al. [**6**], the neural network architecture from NeRF is augmented to accept additional inputs such as time or light coordinates. By training this network using a collection of images with time and light coordinate labels, it is possible to store a full scene representation along with its animation over time and its behaviour under light variations.

Another take on this representation is to use multiple neural networks, each representing a different surface "patch". The union of these patches can then be interpreted as the full

surface of the represented object. This is explored in AtlasNet by Groueix et al. [**18**] and subsequent work by Deprelle et al. [**12**].

Despite their numerous advantages, neural implicit representations have a few downsides. Notably, as the name suggests, it is not straight-forward to extract an explicit surface from this representation. In general, the problem of finding the zero level-set of a function represented by a neural network is non-trivial. Most work rely on a surface extraction technique called marching cubes [**37**] for downstream tasks that require a mesh representation such as real-time rendering. This method is computationally expensive and results in a mesh with limited resolution. In addition, generating a rendering for this representation such as in NeRF is very slow. On a modern top-of-the-line GPU, a single image can take over a minute to render at full resolution because of the slow ray marching computation that needs to be individually computed for each pixel (Equation 2.2.4). This makes neural implicit representations more difficult to use in certain scenarios such as video games where one needs to render the scene at an interactive frame-rate. In addition, it makes training these neural networks very slow and computationally expensive.

2.3. Unsupervised Learning

Unsupervised (or self-supervised) learning differs from supervised learning in that the data used for training does not require labels. In other words, only the inputs are used for learning. This makes it possible to train models using vastly more data, especially in scenarios where labelled data is costly to acquire. This is the case for inverse graphics applications, where generating a large database of photo-realistic 3D renderings from complex 3D scene descriptions requires very large amounts of compute power due to the use of expensive unbiased rendering algorithms such as path tracing. Not only that, but just the process of creating (diverse) realistic scene descriptions requires many work hours for 3D modeling artists. For these reasons, we would like to find a solution to the inverse graphics that can be trained in an unsupervised fashion, using one of the generative models described below. Namely, in this section we present an overview of two of the most widely-used generative models: autoencoders and generative adversarial networks.

Before moving any further however, it is important to have a discussion on the different meanings of the word *unsupervised*. Indeed, not all unsupervised methods are equal as not all methods described as unsupervised actually use no amount of supervision. This is an unfortunate consequence of paper authors branding their inverse graphics approach as "unsupervised" because it uses one less source of supervision than previous approaches. For example, and referring back to the inverse graphics pipeline diagram in Figure 1.1, some methods that improved upon predecessors by removing the need for direct 3D supervision (images labelled with a full 3D model) were said to be unsupervised but actually still required

camera information supervision like camera position and view direction. There are many other ways that some varying amounts of supervision can be used during the learning process, for example by using multiple views of the same scene during training. It is worth mentioning that there is value in finding ways to remove any amount of supervision for the inverse graphics task, although some *types* of supervision are more technically challenging to remove than others. In fact, most methods to this day still require camera supervision and very few are approaches that are actually fully unsupervised. In this thesis we refer to methods that — while not fully unsupervised — use types of supervision that are either considered minimal or easy to acquire as *weakly supervised*. The state of the art report on neural rendering by Tewari et al. [**65**] compares an extensive list of inverse graphics approaches on the type of supervision they use as well as their inputs, outputs and other useful categorizations.

We refer the reader to the Deep Learning book [**16**] for more information on unsupervised learning and generative models.

2.3.1. Autoencoders

An autoencoder is a simple unsupervised machine learning model. Its architecture consists of an encoder neural network E and a decoder neural network D. The input is passed through the encoder before passing the result to the decoder. The system is trained to learn to reproduce a given input according to the following objective:

$$\min_{E,D} \mathbb{E}\left[\mathcal{L}(\mathbf{x}, D(E(\mathbf{x})))\right] \tag{2.3.1}$$

The encoder E takes as input $\mathbf{x}$ and its output is passed as input to the decoder D. $\mathcal{L}$ is the reconstruction loss we want to minimize and can for example be the MSE loss or – if the input/output values are in the range [0,1] – the cross entropy loss.

Typically the size of the output of E is smaller than that of its input $\mathbf{x}$. This creates an information bottleneck which is not only useful for compression, but also for learning meaningful representations.

A *variational autoencoder* (VAE) is a version of an autoencoder introduced by Kingma and Welling [**31**] that better models the training set distribution and makes it possible to generate new, unseen samples. The idea is that, in order to turn the autoencoder into a generative model, we would like to be able to take random samples in the latent space of the autoencoder and pass them through the decoder network to generate new data. Nevertheless, in a regular autoencoder we do not have a mechanism for sampling the latent space. The solution introduced in VAEs is to have the encoder output a distribution from which a random sample is taken before passing through the decoder. This distribution is usually chosen to be a diagonal Gaussian model, meaning that for each element in the latent space the encoder needs to output two parameters: the mean μ and the standard deviation σ.

Then, to be able to generate unseen samples, a regularizer is added to the loss function that encourages the latent space to match a reference distribution. This reference distribution can be sampled at test time to generate novel samples. The new objective is thus of the form:

$$\min_{E,D} \mathbb{E}\left[\mathcal{L}(\mathbf{x}, D(E(\mathbf{x}))) - \lambda D_{\mathrm{KL}}(E(\mathbf{x})||\mathcal{P})\right] \tag{2.3.2}$$

D_{KL} is the Kullback–Leibler (KL) divergence, a formula for computing the difference between a given distribution and a reference distribution:

$$D_{\mathrm{KL}}(p(\mathbf{z})||q(\mathbf{z})) = \int q(\mathbf{z}) \log\left(\frac{q(\mathbf{z})}{p(\mathbf{z})}\right) d\mathbf{z} \tag{2.3.3}$$

where λ is a scalar hyperparameter used to control the importance of the KL divergence term. $\mathcal{P}$ is the distribution of the latent space and is usually chosen to be a Gaussian distribution $\mathcal{N}$ with mean 0 and variance 1. In this case, we want the encoder to also output a normal distribution, defined by μ and σ. The KL divergence term can then be computed analytically:

$$D_{\mathrm{KL}}(\mathcal{N}(\mu(\mathbf{x}), \sigma(\mathbf{x}))||\mathcal{N}(0,1)) = \frac{1}{2}\mathbb{E}\left[\sigma(\mathbf{x}) + \mu(\mathbf{x})^2 - 1 - \log(\sigma(\mathbf{x}))\right] \tag{2.3.4}$$

Additionally, the process of sampling the Gaussian distribution $\mathcal{N}(\mu(\mathbf{x}), \sigma(\mathbf{x}))$ can be done differentiably using the "reparameterization trick". This consists of taking a random sample from the unit Gaussian distribution $\mathcal{N}(0,1)$, then multiplying it by $\sigma(\mathbf{x})$ before adding $\mu(\mathbf{x})$. This allows backpropagating the gradients through μ and σ while still getting unbiased samples.

Autoencoders can be useful for representation learning, but also other tasks such as compression and denoising. They are also applicable in various settings, for example in computer vision where the encoder and decoder models are typically convolutional neural networks such as U-Net [**57**] that progressively shrink the feature map sizes through pooling layers in the encoder before progressively re-expanding them back to the original image resolution in the decoder. In the context of inverse graphics, one can see the training pipeline illustrated in Figure 1.1 as a type of autoencoder where the decoder is a differentiable renderer and where the hidden representation consists of a scene description and is thus easily interpretable.

2.3.2. Generative Adversarial Networks

Generative adversarial networks (GANs) [**17**] are another type of generative model that has been shown to generate sharper images than VAEs which often suffer from blurriness caused by the choice of loss function. This type of model relies on an adversarial game between a *generator* and *discriminator*. The discriminator is trained to distinguish between

real and fake (generated) data points. At the same time, the generator is trained to produce data that fools the discriminator from random samples of a reference distribution. Therefore, as the discriminator gets better at differentiating between real and fake data, the generator also has to keep improving the quality of its output in order to deceive the discriminator. This adversarial game can be formalized mathematically as the following minimax objective function:

$$\min_G \max_D \mathop{\mathbb{E}}_{\mathbf{x}\sim\mathbb{P}_r} [\log(D(\mathbf{x}))] + \mathop{\mathbb{E}}_{\hat{\mathbf{x}}\sim\mathbb{P}_g} [\log(1 - D(\hat{\mathbf{x}})] \tag{2.3.5}$$

where G is the generator, D is the discriminator, $\mathbf{x}$ is a data point from the training dataset $\mathbb{P}_r$ and $\hat{\mathbf{x}} = G(\mathbf{z})$ is a sample created by the generator using a random sample $\mathbf{z}$ from a reference distribution $p(\mathbf{z})$. In practice, this objective is optimized by alternating updates to the generator and to the discriminator networks. The following two objectives are thus optimized in alternation:

$$\max_D \mathop{\mathbb{E}}_{\mathbf{x}\sim\mathbb{P}_r} [\log(D(\mathbf{x}))] + \mathop{\mathbb{E}}_{\hat{\mathbf{x}}\sim\mathbb{P}_g} [\log(1 - D(\hat{\mathbf{x}})] \tag{2.3.6}$$

$$\min_G \mathop{\mathbb{E}}_{\hat{\mathbf{x}}\sim\mathbb{P}_g} [\log(1 - D(\hat{\mathbf{x}}))] \tag{2.3.7}$$

GANs are not perfect, however, as they suffer from instability issues due to the careful balance that needs to exist between the discriminator and generator networks. Additionally, GANs can suffer from *mode collapse* where the generator learns to model only a subset of the true data distribution and ignores other, perhaps more difficult to represent, data points.

To overcome mode collapse, Arjovsky et al. introduce WGAN [**2**] in which they derive a new training objective based on Earth-Mover's distance. The new objective is as follows:

$$\min_G \max_{D\in\mathcal{D}} \mathop{\mathbb{E}}_{\mathbf{x}\sim\mathbb{P}_r} [D(\mathbf{x})] - \mathop{\mathbb{E}}_{\hat{\mathbf{x}}\sim\mathbb{P}_g} [D(\hat{\mathbf{x}})] \tag{2.3.8}$$

where $\mathcal{D}$ is the set of 1-Lipschitz functions. To ensure this, the authors propose to clip the weights of the discriminator to be within some compact space $[-c, c]$.

Nevertheless, this weight clipping approach can lead to vanishing gradient problems, as well as a skewed distribution of weights in the discriminator. For this reason, Gulrajani et al. propose WGAN-GP [**19**] in which gradient clipping is removed in favour of a gradient penalty term that is added to the objective:

$$\lambda \mathop{\mathbb{E}}_{\tilde{\mathbf{x}}\sim\mathbb{P}_{\tilde{\mathbf{x}}}} \left[(||\nabla_{\tilde{\mathbf{x}}} D(\tilde{\mathbf{x}})||_2 - 1)^2 \right] \tag{2.3.9}$$

where λ is a scalar hyperparameter used to control the impact of the gradient penalty term. $\mathbb{P}_{\tilde{\mathbf{x}}}$ is the set of points along the line segment connecting $\mathbf{x}$ and $\hat{\mathbf{x}}$. The sample $\tilde{\mathbf{x}}$ is taken with uniform probability.

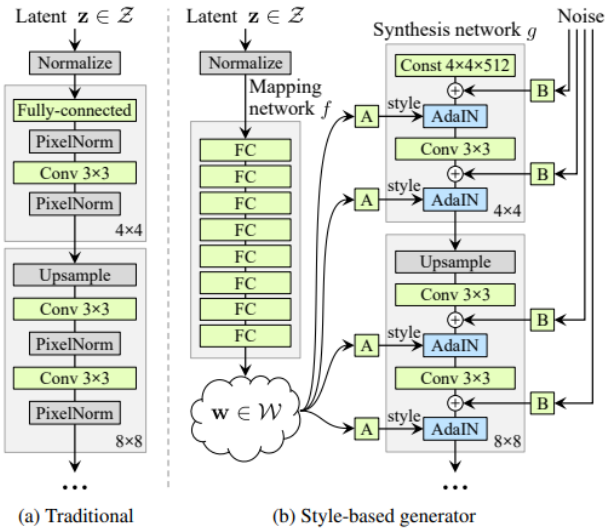

Fig. 2.11. Comparison of a traditional GAN architecture (a) with the approach proposed in StyleGAN (b). The latent code $\mathbf{z}$ is first passed through a series of fully connected layers FC before being introduced in the generator at each upsampling stage through AdaIN normalization. The block A is a learned affine transformation and the block B is a learned per-channel scaling factor. Image taken from StyleGAN [**27**].

Compared to VAEs, GANs do not have an efficient mechanism for inferring a latent code from a data point and thus cannot be directly used for applications such as photo editing. A solution to this is proposed in two concurrent works: Adversarially Learned Inference (ALI) [**14**] and Adversarial Feature Learning (Bi-GAN) [**13**]. The idea is to add an encoder network that takes a data point $\mathbf{x}$ as input and outputs its latent representation $\mathbf{z}$. The discriminator is also modified to take the latent representation $\mathbf{z}$ as an additional input. The discriminator is therefore now tasked with determining whether a latent variable and data point *pair*. Only in the case of a real data point (from the dataset) is the latent variable inferred through the encoder network. This type of model can be trained with an additional reconstruction loss similarly to how autoencoders are trained.

In StyleGAN, Karras et al. [**27**] propose a GAN architecture based on ideas from style transfer approaches to produce high quality image outputs. Instead of using a traditional architecture where the latent code is mapped to a small 2D feature before being upsampled in multiple stages to reach the output resolution, their approach upsamples a learned constant 2D feature. The latent code is instead introduced through Adaptive Instance Normalization (AdaIN) blocks, an idea from style transfer literature. This work builds on Progressive GAN [**26**] and shares the idea of training the generator to output images at increasingly higher resolutions, starting from something small like 4×4 or 8×8 and doubling at each step by adding an upsampling+convolution block until reaching the desired output resolution.

The full architecture of StyleGAN and comparison to traditional approaches are shown in Figure 2.11.

One other aspect of GANs that is important to mention is the difficulty of finding good performance metrics to evaluate and monitor them. Indeed, as the data generated cannot be directly compared to the training dataset, and since the generator and discriminator performance keeps shifting as both models train, it is non-trivial to get meaningful performance scores. To address this problem, Salimans et al. propose the *Inception score* [**59**] where a pretrained Inception model is used to evaluate the quality of the generated images as well as their diversity. Heusel et al. propose the *Fréchet Inception Distance* [**20**] where, as an improvement over the Inception score, the statistics of the real data are compared to the statistic of the synthetic samples.

In this work we explore how unsupervised learning, differentiable rendering and the choice of 3D representation can be combined to address the problem of unsupervised inverse graphics. Over the next two chapters we describe two novel methods using different 3D representations (surfels and a new Voronoi-based representation), differentiable rendering algorithms and unsupervised learning architectures (GAN and autoencoder).

Chapter 3

Unsupervised Depth Estimation from Natural Images

3.1. Introduction

While the goal of inverse graphics is to obtain an explicit 3D representation of a scene from a single natural image, the explicit nature of the representation is not always required for downstream tasks. Using an implicit representation in an architecture designed with a specific task in mind can simplify the challenge at hand, at the cost of adaptability to other applications. Inferring such a representation for a single natural image can therefore be seen as a sub-problem of inverse graphics. If one can solve this problem, others can then think about how to either 1) extract an explicit 3D representation out of the implicit one, or 2) improve the architecture to directly output an explicit representation.

In this work, we explore a novel model architecture to perform both novel-view synthesis and monocular depth estimation in a fully unsupervised manner. Our approach represents the full scene implicitly as a latent code and the weights of a neural network which is decoded for a given camera pose into an explicit representation (surfels) of the visible portion of the scene. This mixture of implicit and explicit representations allows us to directly obtain a depth map of the scene from a given point of view but makes it difficult to retrieve a full 3D mesh model of the entire scene including occluded or out-of-view parts. We view this goal as a significant step towards unsupervised inverse graphics.

Our approach, inspired by previous work, attempts to disentangle a scene's identity from its pose by forcing the network at training time to generate an image from a representation that has been transformed through a random rigid-body transformation. Unfortunately, while our approach is able to generate good novel views on a synthetic dataset consisting of many images of the same scene, it fails at generating reasonable depth maps and novel views on a more complex dataset of natural images of bedrooms. We explore possible explanations

for the failure modes of our approach and compare our work to the successes of concurrent approaches.

In this chapter, we begin by situating our work in the context of previous works in Section 3.2. We then present our method in Section 3.3, followed by our results in Section 3.4. We conclude with a short discussion and conclusion of our findings in Section 3.5.

3.2. Preliminaries

Before delving into the project idea and method, we present where it comes from and position it in the context of the state of the art in unsupervised inverse graphics (at the time). This project idea stems from a couple of previous work, namely *Pix2Shape* [**56**] and *HoloGAN* [**44**]. In this section we provide a detailed view of both of these papers. The techniques used in these work and explained here are used as a starting point for us to build and improve upon and are thus critical to understanding our contribution.

3.2.1. Pix2Shape

In Pix2Shape, Rajeswar et al. [**56**] propose a novel approach towards unsupervised 3D scene understanding from a single image. Their method is based on a neural network that is given a latent representation of a scene and a camera viewpoint to learn to output a surfel representation of the visible part of scene from this viewpoint. This approach can be categorized as weakly supervised since, although it does not require any 3D geometry supervision, it requires the lighting and materials in the scene to be fixed and known as well as the camera position to be given. Additionally, this method relies on the assumption that the world is composed of piece-wise smooth 3D elements. Given all these limitations, the authors only test their approach on synthetic scenes and leave a potential extension of this work for natural images to future work.

Nonetheless, their method was a first step in the direction of 3D scene understanding from natural 2D images. It showed that it is possible to leverage prior knowledge about the image formation process by using a differentiably renderer to tackle the task of unsupervised single image 3D understanding. Simply from lighting queues coming from the differentiable renderer is the neural network able to learn an accurate 3D scene representation in order to perform novel view synthesis and depth estimation from a single image.

The Pix2Shape approach is summarized in Figure 3.1. The architecture employs a GAN with an inference mechanism based on ALI [**14**]. An input image can therefore be passed to the encoder network from which a latent code is obtained. At test time, this latent code can be passed to the decoder (generator) network and differentiable renderer along with a camera viewpoint in order to obtain a novel view of the scene inferred from the input image. This model is trained end-to-end on a synthetic dataset consisting of pairs of viewpoints

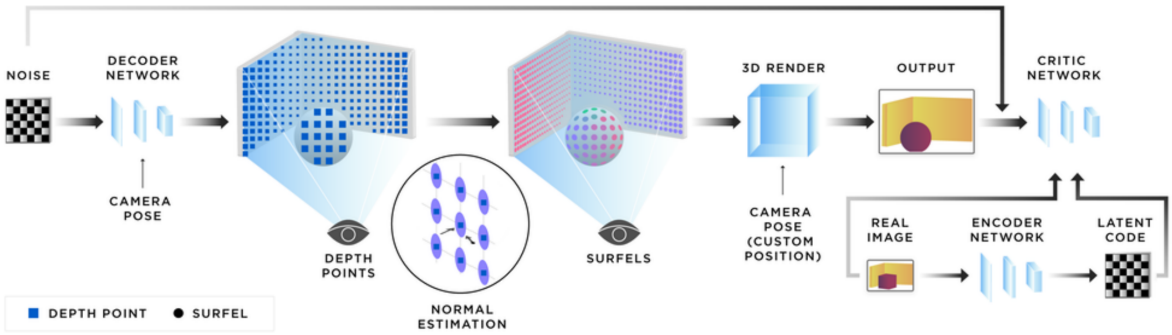

Fig. 3.1. Overview of the training setup of Pix2Shape. The model is based on the ALI framework [**14**] for adversarial inference and is trained end-to-end using a dataset of images with camera pose labels. Image from Pix2Shape [**56**].

and images rendered from randomly generated scenes (without ever taking multiple images from the same scene). These scenes consist of a room with random primitive shapes. A second dataset generated using random ShapeNet [**7**] objects is also used to test the model on more complicated geometry. The lighting and material properties for all scenes is fixed and known.

The decoder is conditioned on the camera pose and generates an explicit surfels representation of the *visible* part of scene from a given viewpoint. Surfels consist of a triple $(\mathbf{p}, \mathbf{n}, \rho)$ where $\mathbf{p}$ is the world position, $\mathbf{n}$ is the surface normal and ρ is the albedo of the material, chosen to be diffuse. The surfels are generated in the camera coordinate system meaning that only one surfel needs to be generated per pixel in the output image. Also, since the material properties are constant and fixed throughout the scene, the decoder only needs to output a depth value for each pixel (a depth map). From these values the position $\mathbf{p}$ can be computed by taking the points along camera rays for each pixel at the distance specified in the depth map. Additionally, surface normals $\mathbf{n}$ can be estimated using a first order approximation, leveraging the assumption that surfaces are locally planar. This way of estimating surface normals also removes a potential source of ambiguity since otherwise a normal could be learned for each surfel that perfectly replicates the target image but that is inconsistent with the output depth map.

The differentiable renderer takes the surfels representation, the camera pose and the scene lighting as inputs and produces a rendering of the scene using a differentiable approximation of the rendering equation (2.1.1). Since there is exactly one surfel per pixel in the output image, each pixel can be rendered individually and in parallel, making this process really fast.

This model has several possible applications. As previously mentioned, it can be used for novel view synthesis, but also for 1) manifold exploration by taking random samples in the latent space, 2) image interpolation in the scene domain by interpolating between the latent

codes inferred from each image, 3) conditional scene generation by training, for example, a model using the ShapeNet dataset [**7**] and conditioning the decoder on the shape class, and 4) 3D scene understanding as demonstrated by the authors on a novel task akin to IQ tests named 3D-IQTT in which the model has to correctly classify which of three images corresponds to a rotated version of the object shown in a separate image.

Interestingly, in their work the (full) scene representation is actually the latent code, which, when combined with a trained decoder network, can be used to visualize the environment from any viewpoint. Surfels are only used as an intermediate representation for the purpose of rendering the scene in a physically accurate fashion. This type of implicit representation has the downside that the output view of the scene is not guaranteed to be consistent between different views if the decoder network has not properly disentangled the scene identity and camera pose.

Also of particular interest, a by-product of this method is an efficient depth estimation method. Indeed, since the decoder effectively outputs a depth map, this model can be used to perform single-image depth estimation (and compared to competing approaches on this task), and can perform it in a weakly supervised setting, without the use of ground truth depth maps or LiDAR point clouds as is often used in other work.

3.2.2. HoloGAN

In HoloGAN, Nguyen-Phuoc et al. [**44**] introduce the first fully unsupervised 3D scene understanding generative model that works on natural images. Their approach is able to generate novel views of a scene inferred from a single RGB image. The model is trained in an end-to-end manner on a dataset of natural images of a certain object/shape. The authors did not test their approach on a training set consisting of images from multiple different object classes. Moreover, the models trained in their work only support camera poses that lie on the surface of a spherical cap and that always look at the center of the object. Therefore, for the models tested in this paper, it is not possible to generate novel views from arbitrary viewpoints.

Despite these limitations, their work significantly pushed the boundaries of what was possible to learn from a collection of unlabelled images of a certain type of object. This is especially true since the model successfully learned a 3D understanding for these objects without having access to multiple images of the exact same item viewed from different viewpoints. Where a human would typically go through multiple steps of reasoning to identify correspondences between images and disentangle the style from the shape, this model learned everything implicitly in the weights of a single neural network.

Figure 3.2 shows the overall architecture of the HoloGAN approach. This is a generative model without an inference mechanism that is based on the architecture of a popular and

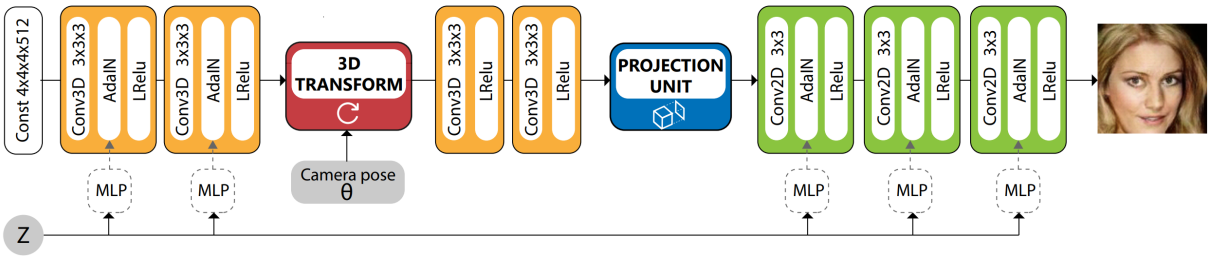

Fig. 3.2. Model architecture of HoloGAN. This generative model is based on the StyleGAN [**27**] where the latent code is introduced through AdaIN layers similarly to style transfer approaches. A voxel representation is first learned before being randomly rotated and projected down to 2D, from which a final image is generated. Image from HoloGAN [**44**].

effective GAN work: StyleGAN [**27**]. This means that the latent code $\mathbf{z}$ is introduced through the use of adaptive instance normalization (AdaIN) layers at various layers in the model. The architecture thus consists of a learned constant tensor of size $4 \times 4 \times 4 \times 512$ followed by a series of upscaling 3D convolution with AdaIN blocks before going through a rigid-body transformation block. The resulting rotated 3D representation is then passed through a couple more 3D convolution layers to allow the model to learn perspective transformation (enabling the use of images taken using different cameras and lenses). After that a learned projection unit is employed to reduce the dimensionality of the hidden representation by one. Finally, a series of 2D convolution blocks with AdaIN layers are used to generate the final image.

As mentioned before, the only rigid-body transformation tested in this work is the rotation around the origin although this process should be applicable to other transformations such as translation. The rotation is achieved by rotating the voxel grid (which at this state has reached size $16 \times 16 \times 16 \times 64$) before performing trilinear resampling to get a new axis-aligned grid. Note how the size of the voxel grid is limited (due to memory and computation constraints of voxel grids). This limits the capacity of this approach to model fine details prior to the 2D convolutional layers. The rotation parameters are randomly sampled within a range that is hand-picked for each dataset. For example, using CelebA, a dataset of human faces, the azimuth rotation range is chosen to be [220,320] degrees and the elevation range [60,95] because there are no images of the backside or top of heads in this dataset. This random rotation transformation, which is applied at training time, is what pushes the network to disentangle between pose and identity as the model needs to learn a representation that will result in an image that will fool the discriminator from any possible viewpoint.

Instead of using a differentiable renderer, this approach learns the rendering mechanism implicitly through the neural network. This allows it to model arbitrarily complex scenes and camera models. In particular, the differentiable projection unit is designed to project a 3D (4D tensor) representation down to 2D while reasoning about occlusion. This is achieved

Fig. 3.3. Comparison of two different views of the same scene using a HoloGAN model trained on the LSUN bedrooms dataset. The top row shows scenes at an elevation of 95 degrees and directly below are the same scenes at an elevation of 60 degrees. Note how some details in the rooms such as the paintings and frames on the wall change between the two views. In some cases such as in the left-most scene, more drastic changes can be seen such as a change in the bed duvet cover.

by using a reshaping layer that takes the depth and channel dimensions of the 4D tensor and merges them together, combined with a few fully-connected layers with non-linearities.

HoloGAN, while impressive, has a few limitations on top of the fact that it only works with camera rotations and that a different model needs to be trained for each type of object. Namely, it is limited by the variety and distribution of poses in the training dataset. It also does not disentangle object appearance from lighting. Additionally, it is not possible to extract an explicit 3D representation out of this model which greatly limits the potential downstream applications of this technique. Finally, the method is limited by the quality of images that generative adversarial networks can generate. The model also does not perfectly disentangle identity and style from camera pose as this exhibits similar consistency issues to Pix2Shape where changing the viewpoint for a fixed scene latent code. Figure 3.3 shows an example of this when using HoloGAN.

It is interesting to note that the choice of using a GAN in HoloGAN is what made it possible for this method to be fully unsupervised. Indeed, as GANs do not require direct comparisons between real and generated images, one can generate images using random camera poses without worrying about having a corresponding target image to compare to.

3.3. Method

Inspired by the success of both Pix2Shape and HoloGAN, we explore a new method for unsupervised inverse graphics. We borrow the idea of using surfels as an intermediate 3D representation from Pix2Shape, but set a more ambitious goal of learning using a fully

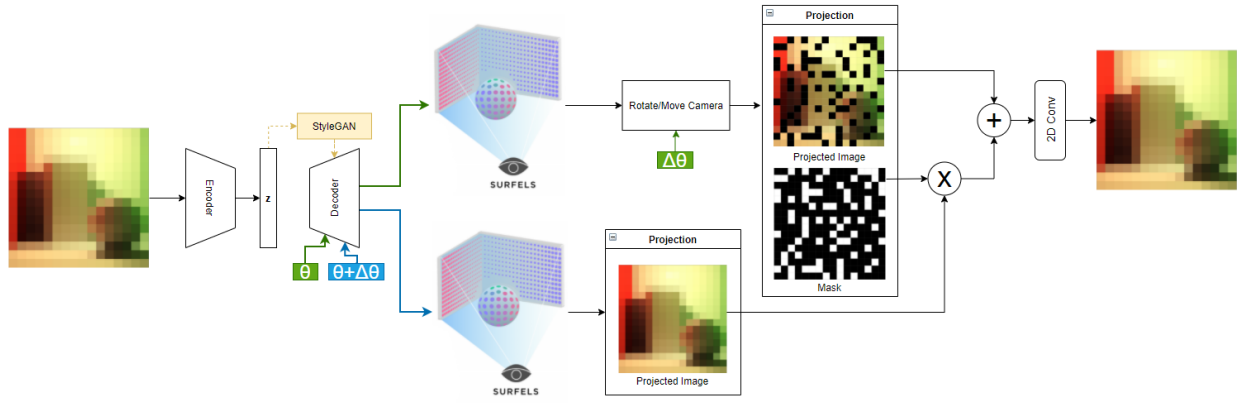

Fig. 3.4. Overview of the proposed architecture for unsupervised inverse graphics, inspired by the approaches in Pix2Shape and HoloGAN. At test time, only a portion of this setup is used: an image is passed as input to the encoder and the decoder is given a camera pose in order to generate an output RGB-D image.

unsupervised setup and with a dataset consisting of natural images. Additionally, our architecture is heavily inspired by HoloGAN and its rigid body transformation layer, however we hope to achieve better view-consistency by using an explicit 3D representation such as surfels. Our core idea is that by using surfels as an intermediate representation we can provide the network with a strong inductive bias about the world through a differentiable surfel rotation and rendering (projection) layer. Such a representation would also allow us to more easily extract 3D information from the architecture such as depth maps, thus making this method suitable for unsupervised monocular depth estimation, another unsolved problem.

3.3.1. Architecture

A diagram of our architecture is shown in Figure 3.4. Similarly to Pix2Shape, we employ the ALI framework for inference in a GAN architecture. At test time and when generating the image and latent code pair for a real dataset sample, an input image is thus passed as input to a CNN encoder from which a latent code $\mathbf{z}$ is obtained. This code is a representation for the full scene and can be decoded into images from different viewpoints using the decoder. This network is also convolutional and is conditioned (using Conditional Normalization [**43**]) on a camera pose θ to output visible surfels (one per pixel like in Pix2Shape) as an N-dimensional image where the first dimension represents a depth map and the other $N-1$ dimensions contain all necessary information for shading the surfels from any given point of view.

During training, in order to disentangle the camera pose from scene identity, we employ HoloGAN's idea of applying random rotations to an intermediate 3D representation before rendering it and passing it through the discriminator. Similarly to Pix2Shape, we compute surfel positions from the decoded depth map and the camera pose before applying the (naturally differentiable) rigid-body transformation that is rotation to the point cloud.

We uniformly sample the camera pose θ from a range corresponding to the chosen dataset and uniformly sample a random rotation angle $\Delta\theta$ from a predetermined limited range. We design a differentiable surfel projection layer that takes the 3D surfel representation and performs a projection operation to obtain a 2D representation as seen from a reference camera aligned along the $-z$ axis. The fact that our projection layer is a physical process and is not learned is a key difference between our architecture and HoloGAN. We hypothesize that this difference will greatly help with view-consistency.

Since the decoder outputs only one surfel per pixel and thus represents only the visible portion of the scene, parts of the scene need to be completed after rotating and projecting the surfel representation. Concretely, this means that the output of the projection layer contains some "empty" pixels that need to be filled in. These pixels are identified in a *mask* image containing values between 0 and 1. This mask is chosen not to be binary and instead can be interpreted as a confidence map of the pixel colors in the output of the projection. In order to fill in the missing pixel values from this image, we use the decoder a second time using the same scene representation code $\mathbf{z}$ but using a different camera pose that corresponds to the rotated viewpoint $\theta' = \theta + \Delta\theta$. We opt for this approach instead of an image inpainting architecture as we want to avoid giving the network the opportunity to learn the entire image formation process in the inpainting layers. This second output from the decoder is used to fill the missing pixel values in the output of the surfel projection layer using the following formula:

$$\mathbf{m} \odot \mathrm{P}(\mathrm{R}(I_\theta, d_\theta)) + (1 - \mathbf{m}) \odot I_{\theta'} \tag{3.3.1}$$

where $\mathbf{m}$ is the mask, $\odot$ is the Hadamard product, R is the rotation layer, P is the projection layer, I_θ is the image (of $N-1$ channels) generated by the decoder from viewpoint θ, d_θ is the depth output from that same decoder pass and $I_{\theta'}$ is the image generated by the decoder from viewpoint $\theta' = \theta + \Delta\theta$.

After the two images I_θ and $I_{\theta'}$ have been merged using the mask, we obtain an image of $N-1$ channels where any "empty" pixels have been filled. These $N-1$ dimensions are assumed to contain all necessary information for shading the surfels from any point of view. This information can be stored in an explicitly defined format, such as spherical harmonics or can be stored implicitly by treating it as a hidden representation. When using the former format, an RGB image can be generated analytically while in the latter case we add a few convolutional layers similarly to HoloGAN to generate the final render while modeling view-dependent effects. For scenes consisting only of diffuse surfaces, N can be set to four and the last three channels are interpreted as pixel RGB values, thus removing the need for any additional convolutional layers.

At test time, the decoder is queried only once using the latent code obtained from the encoder and a desired camera pose. The depth is removed from the output and the resulting image of $N-1$ channels is passed to the final few convolutional layers if there are any in order to obtain a final RGB image. This process therefore outputs both a depth map for the input image and a novel view of the inferred scene.

We implement our approach in PyTorch [**49**] for faster training times on GPU.

3.3.2. Differentiable Surfels Projection

We implement two very different differentiable surfel projection techniques and explore which is better suited for our task and presents the smoother loss landscape. The first method is inspired by the differentiable point cloud technique introduced by Insafutdinov et al. [**22**] and relies on projecting each surfel to a Gaussian splat. The second is based on the differentiable image sampling method from Spatial Transformer Networks by Jaderberg et al. [**23**] in which a source image is sampled for each pixel of a warped output image. Because of their opposite techniques, we refer to the differentiable point clouds and differentiable image sampling techniques as the *direct renderer* and the *reverse renderer*, respectively.

In the **direct renderer** the first step is to compute the image coordinate $\mathbf{x}'_i$ of each surfel when projected onto the camera plane (after rotation). We use a perspective projection matrix for this step. Note that these coordinates can exceed the image bounds, meaning that these surfels are no longer in view after the rotation transformation. The projected surfels cannot be directly rendered as individual pixels at their image coordinates as this would create edge discontinuities similar to traditional triangle mesh rasterization. In a traditional computer graphics point cloud rendering pipeline, the points would also be sorted by depth using a z-buffer which would additionally create occlusion discontinuities. The idea here is therefore similar to differentiable rasterizers such as SoftRas: each projected surfel is smoothened by modeling it as a Gaussian with origin at the surfel's image coordinate. These Gaussian points are semi-transparent which also allows reasoning through occlusions. The final projected value P for a pixel whose center has image coordinate $\mathbf{p}$ is therefore given by:

$$\mathrm{P}(\mathbf{p}) = \sum_{i=1}^{M} I_i \exp\left(-\frac{1}{2}\frac{||\mathbf{p}-\mathbf{x}'_i||_2^2}{\sigma^2}\right) \tag{3.3.2}$$

where I_i is the $(N-1)$-dimensional vector containing the shading information for surfel i, M is the number of surfels and σ is the standard deviation of the Gaussian that determines the amount of blurring applied to each projected surfel.

The mask $\mathbf{m}$ is computed very similarly through the following equation:

$$\mathbf{m}(\mathbf{p}) = \min\left(\sum_{i=1}^{M} \exp\left(-\frac{1}{2}\frac{||\mathbf{p}-\mathbf{x}'_i||_2^2}{\sigma^2}\right), 1\right) \tag{3.3.3}$$

where the output is clipped using a min function to be at most 1.

We implement a more efficient version of this process that scales linearly instead of quadratically with respect to the number of surfels. Instead of computing a sum over all surfels for each output pixel, the projected surfels are first mapped to a cell in the pixel grid using bilinear interpolation. The resulting 2D grid is then blurred using the same Gaussian kernel as in Equation 3.3.2 and blurring is decomposed into two 1D convolutions along each axis individually. When more than one surfel ends up in the same pixel cell, we employ a surfel merging function, of which we explore many different versions.

The first surfel merging function we explore is *simple averaging* where each contributing surfel is weighted equally. This method is the easiest to implement but does not take into account the fact that some surfels within the cell have an image coordinate that is closer to the pixel's center coordinate compared to other surfels. For this reason we explore a second version, *distance-weighted averaging*, where the averaging weights w_i^{dist} are computed using the following equation:

$$w_i^{\text{dist}} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{1}{2}\frac{||\mathbf{p}_i - \mathbf{x}'_i||_2^2}{\sigma^2}\right) \tag{3.3.4}$$

where $\mathbf{p}_i$ is the coordinate of the closest pixel center to $\mathbf{x}'_i$.

Both of these surfel merging functions ignore the effect of occlusion on the final rendering. To rectify this, we design a surfel merging function inspired by fast, order-independent transparency from the field of real-time computer graphics [**40**]. In this *depth-weighted averaging* surfel merging technique the averaging weights w_i^{depth} take the form:

$$w_i^{\text{depth}} = \exp(-\mu * z'_i) \tag{3.3.5}$$

where μ is a scalar controlling the prominence of depth on the weights and z'_i is the depth of surfel i from the camera plane, after rotation. Note that this equation comes from the Beer-Lambert law and that the coefficient μ can thus be interpreted as the extinction coefficient.

Finally, we propose a final surfel merging function which is simply a combination of the distance-weighted averaging and depth-weighted averaging functions. This *dist+depth-weighted averaging* is taken to be simply the multiplication of the weights from Equations 3.3.4 and 3.3.5: $w_i^{\text{dist+depth}} = w_i^{\text{dist}} \odot w_i^{\text{depth}}$.

All weights w_i are normalized to make sure that they sum to 1.

The **reverse renderer** approaches the projection problem from the other direction. Instead of finding where each surfel/pixel in the image I_θ maps to in the rotated output image, it finds the origin of pixels in the output image on the pre-rotation image plane of I_θ. This is done by using a depth map $d_{\theta'}$ from the rotated viewpoint θ', which we already obtain from our approach when using the decoder with viewpoint input θ'. Using the same process as in the direct renderer, we use this depth map to compute world positions for each

surfel/pixel i seen from the output image before applying the *reverse* rotation and projecting to the θ camera plane to obtain image coordinates $\mathbf{x}$. Therefore, for each pixel in the output image, we have a location in the input image I_θ from which we can sample a pixel value. We perform this sampling differentiably by using bilinear interpolation:

$$\mathrm{P}_i = \sum_n^H \sum_m^W I_{nm} \max(0, 1 - |x_i - m|) \max(0, 1 - |y_i - n|) \tag{3.3.6}$$

where H and W are the height and width of the input image $I = I_\theta$ and x_i and y_i are the x and y components of image coordinate vector $\mathbf{x}_i$ for pixel i. In practice, this sampling operation can be implemented efficiently on the GPU by looking only at the four closest neighbours around a given point (instead of the sum over all pixels) and requires only a single line of PyTorch code. Note that this limited region also means that each output pixel is only locally differentiable with respect to those four neighbors in the input image. Although we do not experiment with this, this issue can however be solved by first blurring the input image using a Gaussian kernel.

While this reverse rendering process is simpler than the direct renderer, it is not as straight-forward to compute a reasonable mask that reasons about occlusion. Without considering occlusion, the mask can be trivially computed for each output pixel i by evaluating whether the image coordinate $\mathbf{x}_i$ falls outside or inside of the bounds of the image image. Nevertheless, to factor in occlusion, another step needs to be performed. One way to approximate an occlusion mask would be to compare the depth of close neighbour image coordinates $\mathbf{x}_i$ from viewpoint θ. We instead make this process easier by leveraging the depth map d_θ from viewpoint θ. This way we can compare the depth computed from the projection step z (as seen from viewpoint θ) to the depth generated by the decoder network d_θ. Note, however, that we do not have a direct correspondence between z and d_θ as one is calculated for each pixel in the θ' image and the other for the θ image. To remedy this, we apply the same operation as in Equation 3.3.6 to sample the depth map d_θ using image coordinates $\mathbf{x}$ in order to obtain a depth image $\hat{d_\theta}$ that is aligned with the z image. The mask $\mathbf{m}$ is thus determined as follows for each pixel i:

$$\mathbf{m}_i = (z_i \leq \hat{d}_{\theta i} + \epsilon) \tag{3.3.7}$$

where ϵ is a small scalar to ensure that surfels that have a very similar depth are not marked as occluded in the mask.

This occlusion mask is multiplied with the mask computed by looking at whether image coordinates fall within image bounds to obtain the final mask. Figure 3.5 shows an example set of inputs and outputs (including the mask) of the reverse renderer when given a ground truth depth map.

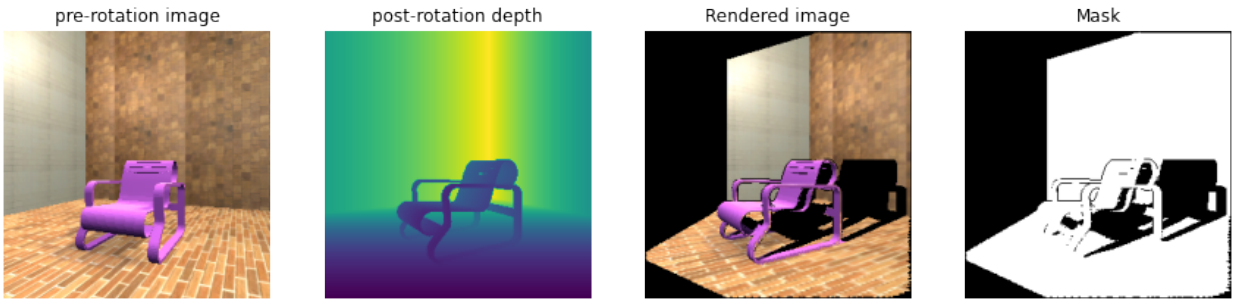

Fig. 3.5. Example inputs and outputs of the reverse renderer. This differentiable renderer takes as input a source image ("pre-rotation image") as well as a depth map from the target rendering viewpoint ("post-rotation depth"). It outputs a rendering of the scene as seen from the target rendering viewpoint by sampling the source image using the input depth map. To account for occlusion and for elements moving in and out of view, a mask is also computed and outputted to indicate which pixels still need to be filled in.

3.3.3. Losses

We train our model adversarially using the ALI framework. The GAN loss $\mathcal{L}_{\text{GAN}}$ is computed as in WGAN-GP [**19**]. In addition, we explore the effect of a couple of regularization losses on the learning process.

We propose to use a **reconstruction loss** $\mathcal{L}_{\text{recon}}$ that compares the output of the differentiable projection layer after rotation by $\Delta\theta$ with the direct output from the decoder conditioned on $\theta + \Delta\theta$ using the MSE loss as follows:

$$\mathcal{L}_{\text{recon}} = \mathcal{L}_{\text{MSE}}\left(\mathbf{m} \odot \text{P}(\text{R}(I_\theta, d_\theta)), \mathbf{m} \odot I_{\theta'}\right) \tag{3.3.8}$$

where the mask $\mathbf{m}$ is used to weigh the loss per pixel. This way the loss is 0 where the mask indicates that there is an "empty" pixel in the projected image.

This loss encourages the network to generate view-consistent images and depth maps.

Additionally, we add a **flattening loss** $\mathcal{L}_{\text{flat}}$ to push the generated depth maps to be locally planar and thus locally smooth. This is achieved by first estimating surface normals $\mathbf{n}$ from the depth map before then comparing each pixel's normal vector with the normals of the ring of eight closest neighbours. The flattening loss is therefore computed as follows:

$$\mathcal{L}_{\text{flat}} = \frac{1}{M}\sum_{i}^{M}\sum_{j\in\mathcal{N}(i)}(-(\mathbf{n}_i \cdot \mathbf{n}_j) + 1)^2 \tag{3.3.9}$$

where $\mathcal{N}(i)$ is the set of eight direct neighbours of pixel i and $\cdot$ is the vector dot product. This dot product computes the cosine of the angle between the two normals $\mathbf{n}_i$ and $\mathbf{n}_j$ which we wish to minimize. Note that the dot product here is guaranteed to return a non-negative value as the normals estimation process assumes that normals are all forward-facing relative to the camera.

The normals are estimated in the same way as in Pix2Shape by computing spatial finite-difference derivatives of the depth map with the eight closest neighbours of each pixel.

All these losses are combined using scalar hyperparameters λ_{recon} and λ_{flat} that help control the impact of each loss on the overall training objective:

$$\mathcal{L} = \mathcal{L}_{\text{GAN}} + \lambda_{\text{recon}} \mathcal{L}_{\text{recon}} + \lambda_{\text{flat}} \mathcal{L}_{\text{flat}} \tag{3.3.10}$$

3.3.4. Datasets

We perform our experiments on two datasets. The first is *LSUN bedrooms* [**72**], a large dataset of 3 million natural images of bedrooms without any associated labels. This dataset allows us to test whether our model has achieved its goal of learning 3D reasoning in a completely unsupervised manner. It also allows us to qualitatively compare our results to those of HoloGAN as they also train a model on this dataset.

We also create a custom synthetic dataset in order to better analyze the learning dynamics of our approach, but also to allow for quicker development and hyperparameter optimization runs. This *chair* dataset serves as an easier test case and proof-of-concept for our model. It contains of 100k images of the same synthetic scene rendered from random viewpoints. This scene consists of a single chair object taken from the ShapeNet dataset [**7**] placed in the middle of a cubic room. The chair is given a wood-like texture while the room is covered in a checkerboard black and white pattern. The viewpoints are sampled on an arc parallel to the ground floor, at a fixed elevation of 45° and therefore only a single axis of rotation is sampled. This arc has a range of ±45° on either side of the canonical view of the chair facing the camera. The entire scene is Lambertian and has no view-dependent effects. This last fact allows us to store for each surfel only 3-dimensional RGB vectors and to remove the final convolutional layers from the architecture shown in Figure 3.4. Additionally, since all images in this dataset are taken from the same scene, we can remove the encoder network and use a fixed latent code $\mathbf{z}$.

3.3.5. Concurrent Work

Concurrently to our work, two other approaches have been proposed which share a lot of similarities to ours.

First, **RGBD-GAN** by Noguchi et al. [**47**] shares our idea of using a view-consistency loss on a decoder that is conditioned on the camera pose. Their approach is similar to HoloGAN in that they train a generative model without an inference mechanism such as ALI and their model can generate images for a given viewpoint. Their conditional decoder outputs a depth map on top of an RGB image for a given camera pose. This depth map is not passed to the discriminator and is instead learned through a 3D consistency loss. This

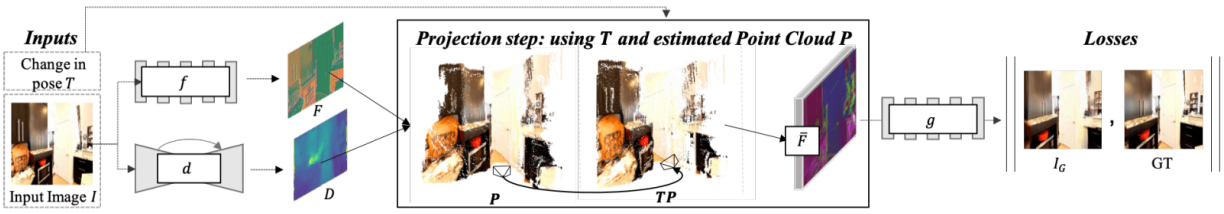

Fig. 3.6. Model architecture of SynSin [**67**].

3D consistency loss is almost identical to our $\mathcal{L}_{\text{recon}}$ loss and uses the same differentiable image sampling technique taken from Spatial Transformer Networks. The only differences between the loss we use and their 3D consistency loss are 1) that we make use of a mask to reason about occlusion whereas they do not compute any mask, and 2) that they not only compare the rotated and projected RGB images but also the rotated and projected depth maps. Similarly to our work, their approach is fully unsupervised and generates explicit depth maps. It is, however, limited to scenes without view-dependent effects and shows limited success at both disentangling identity from pose and at depth map quality. They also do not train with an inference mechanism and thus cannot compute a depth map for a given RGB image and cannot infer a scene from an input image to perform novel-view synthesis.

Second, Wiles et al. propose **SynSin** [**67**], an approach for end-to-end novel view synthesis from a single image that is weakly supervised. They use a dataset of pair of images of the same scene taken from different viewpoints where each image is labelled with the corresponding camera pose. This allows them to use an autoencoder instead of a GAN where the model takes as input one of the two images in a pair and the camera transformation between the two viewpoints of that pair, and is trained to output the second image. Figure 3.6 shows an overview of the model architecture they use. Similarly to our approach, they use as intermediate 3D representation a point cloud that is obtained by predicting a depth map for the given input image. They apply the same rigid-body transformation operation to this representation as we do and use a very similar differentiable rendering process, also based on the paper by Insafutdinov et al. [**22**]. To fill any "empty" pixel as a result of this rotation and projection operation, they opt for an image in-painting approach using a *refinement* convolutional neural network. The biggest difference between our approach and theirs is that we train in a fully unsupervised manner and thus do not require any label or image pair in the datasets we use.

3.4. Results

We first test our approach on the simpler setting of the *chair* dataset to validate our approach and to explore the effect of different hyperparameters, renderers, losses and other

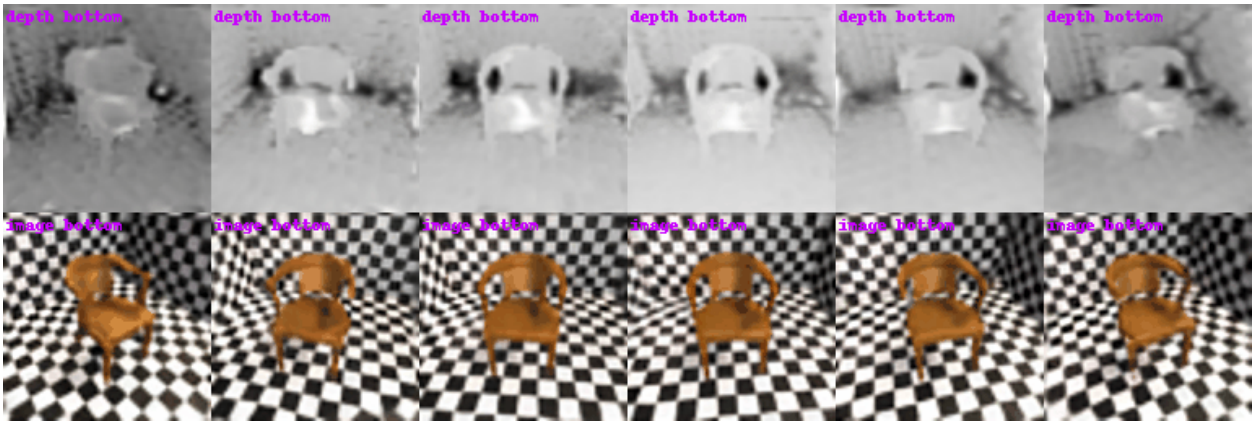

Fig. 3.7. Initial results of the proposed architecture on a synthetic dataset consisting of 100k images of a scene of a chair in a room viewed from random angles. The top row shows the depth (lighter values are closer) while the bottom row shows the RGB image generated from the decoder at increasing azimuthal angles.

tricks. We start by running a version of our model without an encoder for inference. We use a fixed latent code since this dataset consists of different views of the same scene. We thus run a simple GAN with progressive growing as in StyleGAN from a starting resolution of 8x8 pixels all the way up to 128x128 pixels. We also use the mixing regularization loss proposed by StyleGAN. Since the scene in this dataset contains only diffuse surfaces, we model the surfels as a simple RGB-D vector output from the decoder and do not use any final convolutional layers after the differentiable projection layer. We constrain the depth to be within the bounds of the scene by applying and scaling a sigmoid activation function to the depth output. For each sample we generate during training, a random camera azimuth angle θ is sampled uniformly in the range of the dataset, $\pm 45°$. The rotation angle $\Delta\theta$ is also sampled uniformly in a range that increases over training time as a form of curriculum learning. This range starts at $\pm 0°$ and caps at $\pm 30°$ after 150k iterations. We train our model for 3M iterations at each resolution level for a total of 15M iterations, which takes approximately 6 days on a single Nvidia V100 GPU. We use a batch size of 128 at the 8x8 resolution level, followed by progressively smaller batch sizes of 64, 32, 16 and 16 again for the resolutions 16x16, 32x32, 64x64 and 128x128, respectively. We train using the Adam optimizer with a learning rate of $1e^{-3}$. In this first experiment we use the *forward renderer* along with the *simple averaging* pixel merging function. We progressively decay the blurring parameter σ of the forward renderer starting from 0.14 with decay factor of 0.25 per epoch to generate progressively sharper images over training time. Finally, we first train without any regularization losses and thus set λ_{recon} and λ_{flat} to 0.

The results of this experiment can be seen in Figure 3.7. We only perform a qualitative analysis of these results as there exist no good metrics for determining whether the identity and pose are well disentangled by the model. It is also not possible to get a ground truth

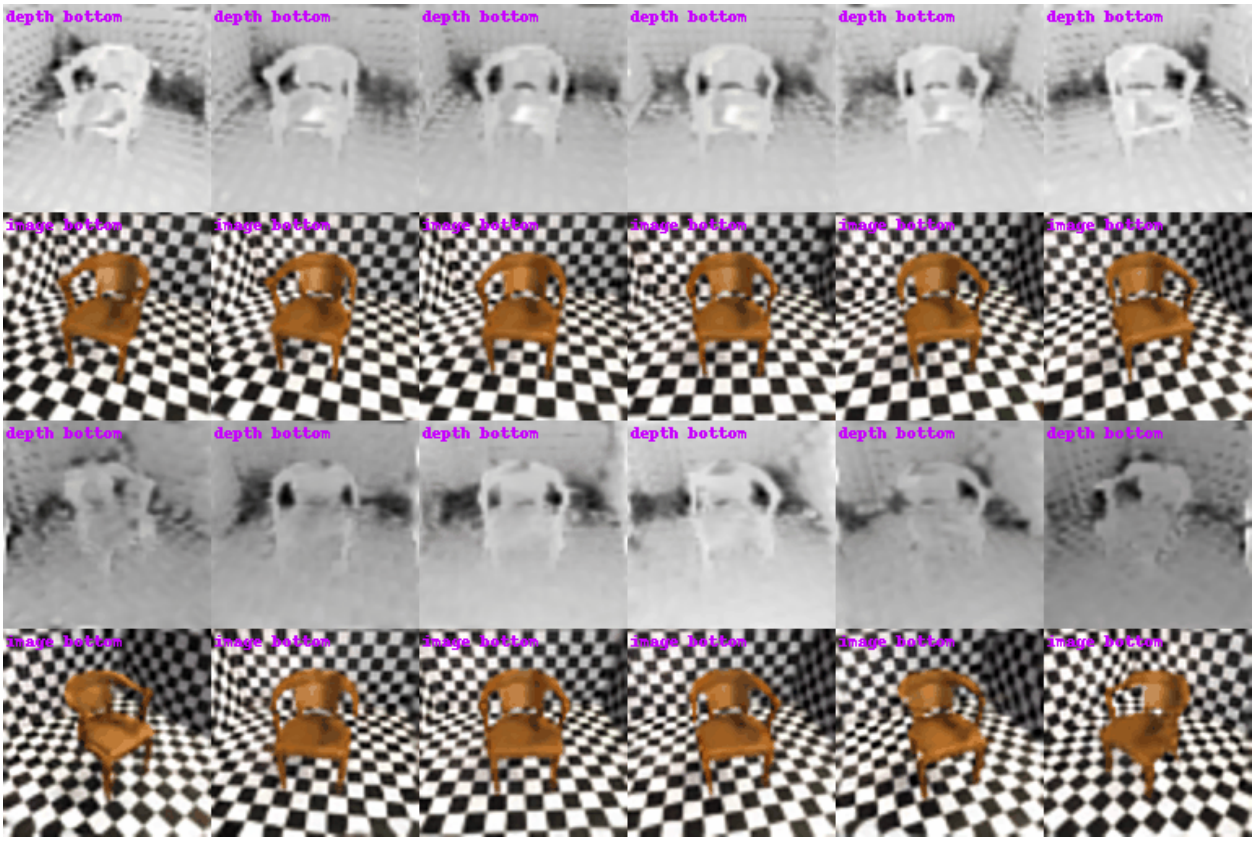

Fig. 3.8. Results of adding a flattening loss. The first two rows show the result when using a weight of 0.5 while the last two rows show the result with a weight of 5.

depth map for an image generated by our approach since the angle passed as conditioning for the decoder does not necessarily match the reference angle. The results of Figure 3.7 show that our model is able to generate images of the scene from various viewpoints, although the left-most image shows a discontinuity in the θ range. While the generated RGB images look reasonable, the generated depth maps show some issues. The depth does not smoothly increase on the walls and floor of the room from the camera to the back of the room. Furthermore, some arbitrary regions on the flat surfaces of the chair have a different depth than the rest of those surfaces. Since the RGB images generated are fine despite these flaws, this indicates a potentially big issue not only for the task of monocular depth estimation, but also for the application of our technique to datasets of natural images in general since it indicates that there is ambiguity between the depth maps and the generated images. In other words, there are many possible depth maps that can be used to generate the same output image.

Following this first experiment we explore some other techniques and tricks to try to improve our initial results and to remove the depth map ambiguity.

First, we run the same experiment but add the flattening loss $\mathcal{L}_{\text{flat}}$ and experiment with a couple values of λ_{flat}: 0.5 and 5 in the hopes of resolving the ambiguity issue. Figure 3.8 shows the same plot of increasing azimuth angle for this experiment. We notice a slight improvement compared to our initial experiment when using this loss with a small scaling factor λ_{flat}. The generated depth maps look better but still exhibit signs of ambiguity.

We then run a series of experiments to compare the different proposed surfel merging techniques. This set of four runs examine the resulting differences in using the *simple averaging*, *distance-weighted averaging*, *depth-weighted averaging* and *dist+depth-weighted averaging* functions. The results, shown in Figure 3.9, demonstrate a significant improvement when using the more involved *dist+depth-weighted averaging* technique. Given this performance improvement, we decide to only use this method moving forward when using the forward renderer.

Finally, we run some experiments using the reverse renderer to compare its performance against the forward renderer. We also use this opportunity to run a couple more experiments to determine the impact of curriculum learning on our architecture. We thus run three experiments using the reverse renderer. The first uses a fixed $\Delta\theta$ range that does not increase over training time. The second uses a progressively increasing $\Delta\theta$ range but does not use the progressive GAN training trick of StyleGAN. The third experiment is a combination of the other two where neither the $\Delta\theta$ range nor the resolution of the generated images change over training time. Results for these three experiments are presented in this order in Figure 3.10. These results show that 1) the reverse renderer results is not only better depth map quality, but also sharper output RGB images, and 2) curriculum learning leads to worse performance for our model. We hypothesize that the progressive training trick of StyleGAN might be negatively affecting our approach as the image and depth map need to have a high enough resolution to provide a meaningful signal for small $\Delta\theta$ rotations. Using the reverse renderer has the additional benefit of having two less hyperparameters as we do not need to worry about controlling the blur factor σ or choosing a surfel merging technique.

We combine these findings to train a new model on the more complex LSUN bedrooms dataset which contains natural images of different bedroom scenes. We do not yet use an encoder and the ALI inference framework. In comparison to our model trained on the chair dataset, we use a random latent code sampled from a normal distribution and set the surfel features' number of dimensions to 32. A single 1×1 convolutional layer is added at the end of our network architecture to perform the final rendering step and to reduce the number of output channels to three to output an RGB image. We use the same resolution and azimuth and elevation angle ranges for this dataset as HoloGAN, namely a resolution of 128×128 pixels, an azimuth range of $220° - 320°$ and an elevation range of $60° - 95°$. Learning from previous experiments, we use a flattening loss scaling λ_{flat} of 0.5, use the reverse renderer instead of the forward renderer and do not use the progressive training trick of StyleGAN.

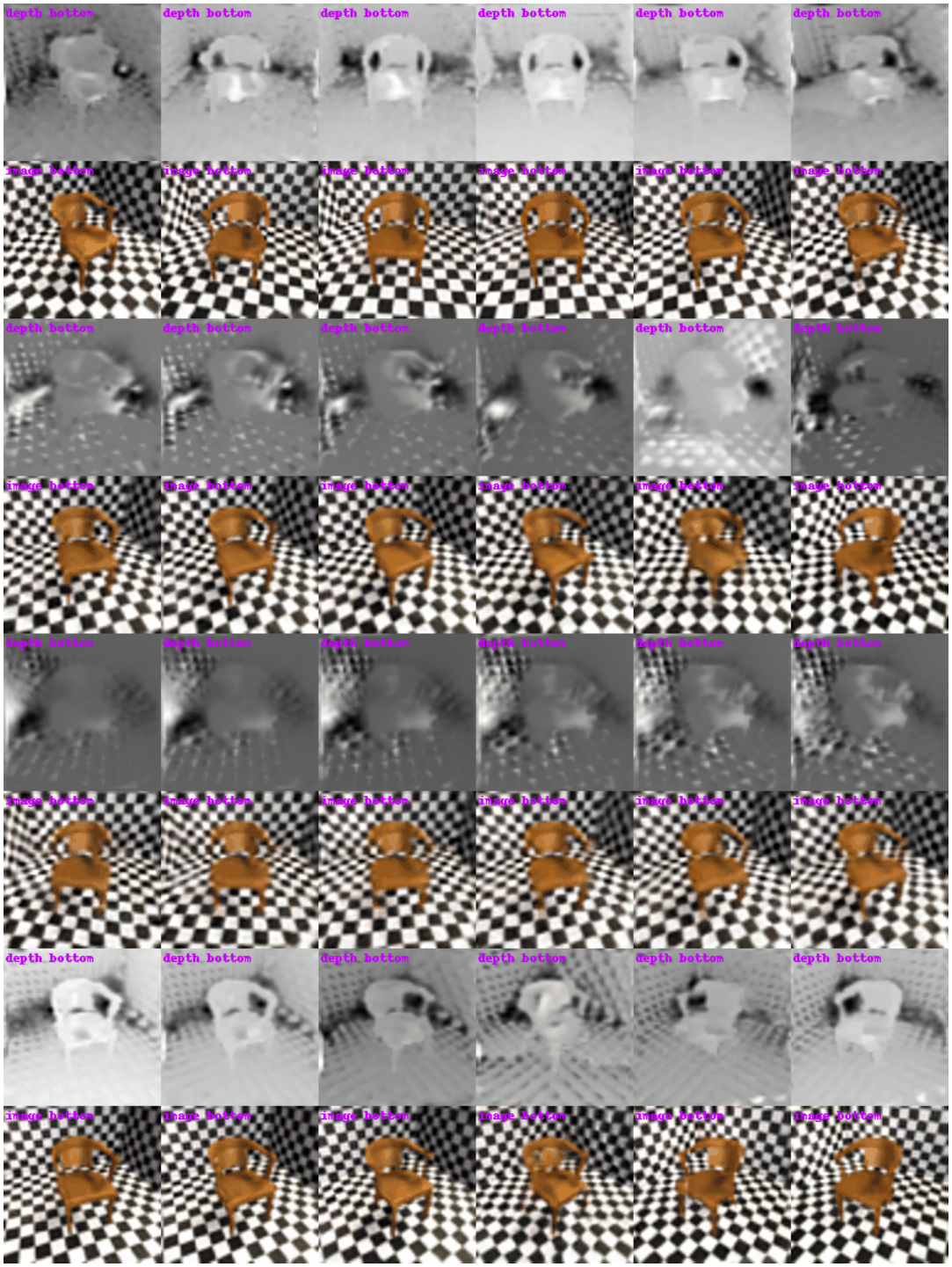

Fig. 3.9. Comparison of different surfel merging techniques. From the top, in groups of two rows, the results are presented in the following order: simple (non-weighted) averaging, weighted averaging using the distance to the closest pixel center, weighted averaging using the depth and finally weighted averaging using a combination of depth and distance to pixel center.

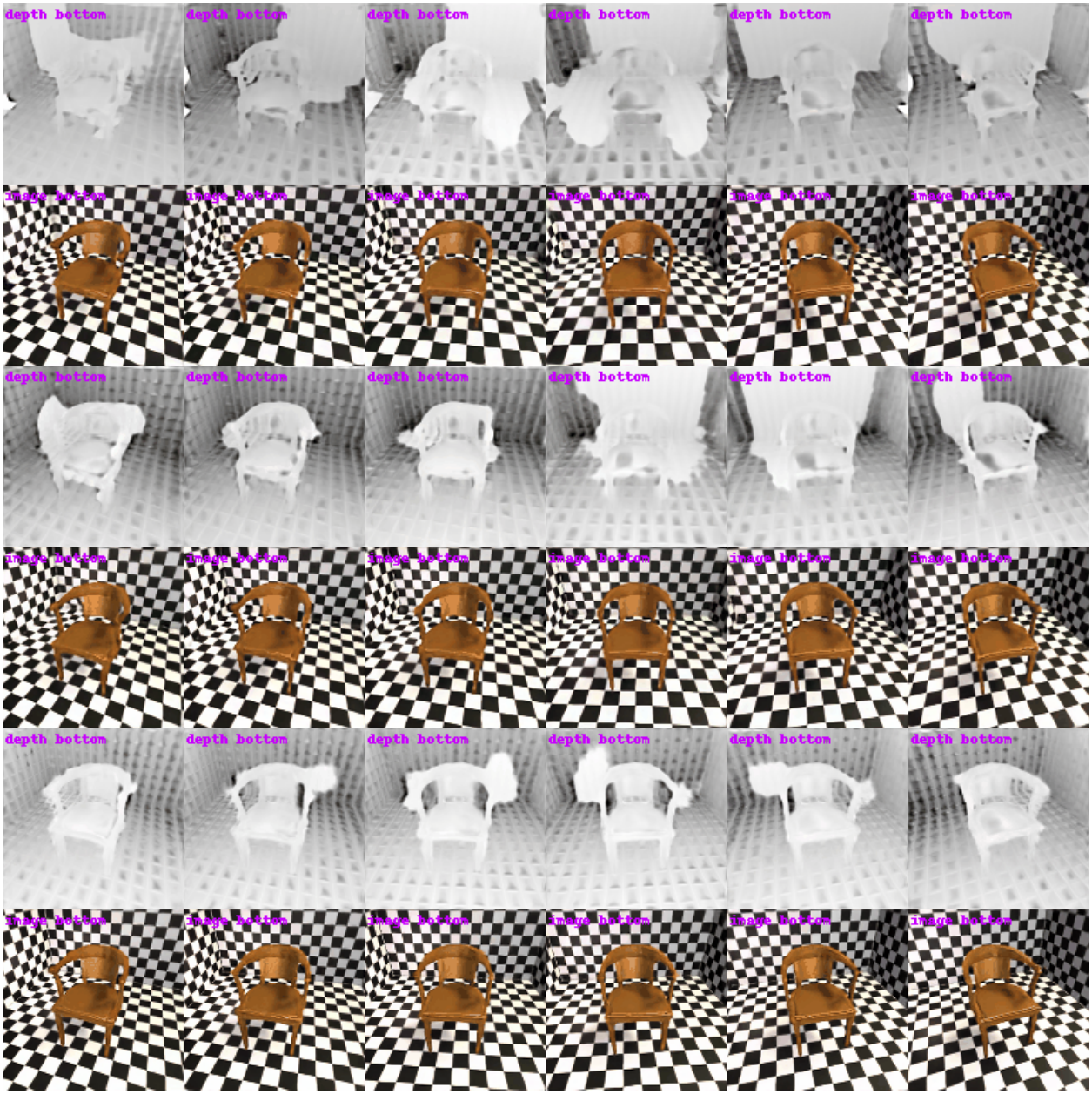

Fig. 3.10. Results using the reverse renderer. The first two rows show a direct comparison to the experiment shown in Figure 3.7. The next two rows show the result using a progressively increase $\Delta\theta$ value over training time and the last two rows show the same experiment but trained without the progressive training idea of StyleGAN.

We run two versions of this experiment, one where the $\Delta\theta$ range progressively increases over training time as in our initial experiment on the chair dataset, and one where this range is fixed throughout training to $\pm 30°$ for the azimuth angle and $\pm 15°$ for the elevation.

Figure 3.11 shows example scenes generated using the two models trained as described above. The first two rows show the generated depth maps and RGB images from the approach that uses a fixed $\Delta\theta$ range. The bottom two row show the other experiment where this range

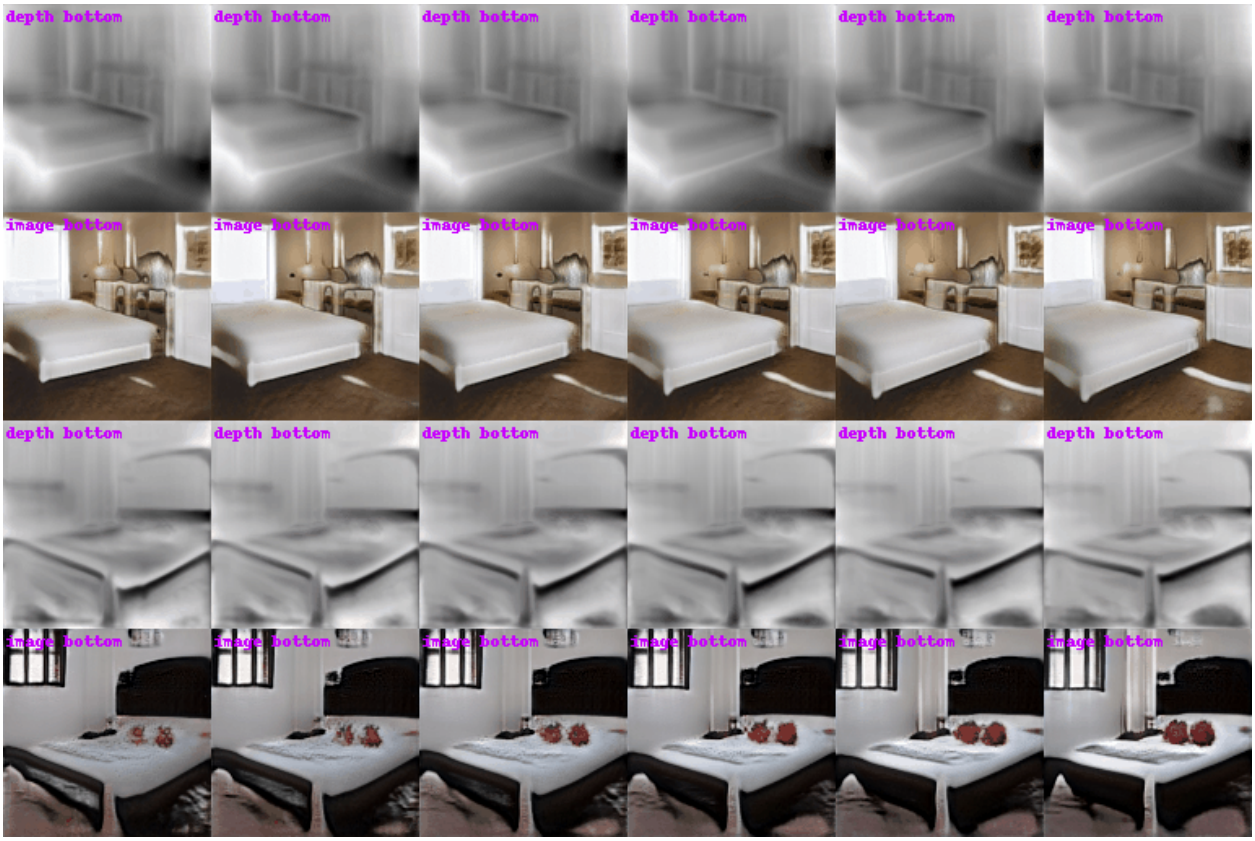

Fig. 3.11. Results of our best hyperparameter configuration on the LSUN bedrooms dataset. The bottom two rows correspond to an experiment that differs from the one in the first two rows in that $\Delta\theta$ is progressively increased throughout training. These results show that, in both cases, the quality of the generated depth map is lacking and that the model seems to have found a way to make depth maps that lead to a reasonable final image quality while limiting the perceived rotation of the scene.

is progressively increased over training time. When looking at the generated depth maps, while we see that the network has a good understanding of object edges, we find that the depth values are very far from realistic. For example, object edges should only barely be visible in these images wherever both surfaces on either side of these edges are visible. Furthermore, we find very little consistency in the depth values on walls and floor of the bedrooms. This poor performance on depth map generation leads to significant issues in the novel-view generation process. We notice in the visualizations of Figure 3.11 that the camera pose does not appear to rotate as much as expected. This is especially obvious when compared to HoloGAN on the same azimuth range of $220° - 320°$, as shown in Figure 3.12. It appears that the depth ambiguity, as observed above, was exploited by the neural network to more easily generate images by limiting the effect of the θ conditioning in the decoder network. The learned depth maps lead to a minimal amount of rotation when passed through the differentiable projection layer.

Fig. 3.12. Example scenes generated by HoloGAN where the scene latent code is kept fixed but the azimuth angle is progressively increased. Note that HoloGAN does not provide a mechanism for obtaining depth maps.

These flaws identified in our results indicate some issues related to the depth map ambiguity and possibly with the smoothness of the differentiable projection layer. For this reason, we run additional experiments on the reverse renderer to determine how effective of a gradient signal it provides to the decoder network given that differentiable renderers are notorious for getting stuck in local minima easily. We render two viewpoints of a synthetic scene consisting of a single pink ShapeNet chair in the middle of a room with random textures on the walls and floor. We note the angle difference between these two viewpoints and additionally render two corresponding depth maps.

This allows us to run an experiment where we directly optimize a depth map image by computing a loss comparing the output of our reverse renderer to the ground truth image; the second viewpoint rendered using our synthetic scene. The reverse renderer is thus given as input a randomly initialized depth map, a source image from the first viewpoint to sample from and the rotation transformation between the two views. The random depth map is then optimized iteratively using stochastic gradient descent from an MSE loss that compares the rendered output to the ground truth image from the target viewpoint. Figure 3.13 shows the final outputted image and learned depth map after 2000 iterations and compares them to the ground truths. We find that this setup is unable to recover the target depth map and thus to generate a reasonable rotated image. In addition, we monitor during training the MSE loss between the learned and target depth maps and find that, while the depth loss increases over training time, the depth barely changes over the course of optimization. Quantitatively, while the image MSE loss decreases by approximately 0.03 the depth MSE loss increases by 0.0006, suggesting that the gradient signal to the depth is too small to be useful.

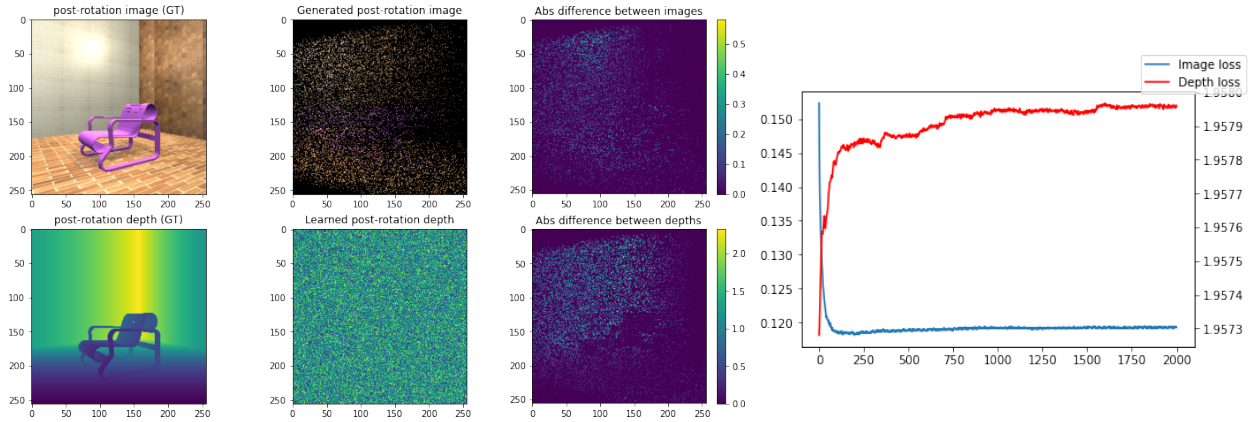

Fig. 3.13. Direct optimization of the depth map through the reverse renderer. On the left, the first row of images compares the ground truth target image with the rendered one while the bottom row compares the ground truth depth map with the learned depth map. On the right we plot the MSE loss between the target and generated images (the training signal) in blue and plot the MSE loss between the ground truth depth and learned depth in red (this loss is not used in training). Note that the image loss and depth loss y-axis scales are shown on the left and right of the plot respectively. In this experiment, the depth map is randomly initialized using uniform sampling in the bounds of the scene.

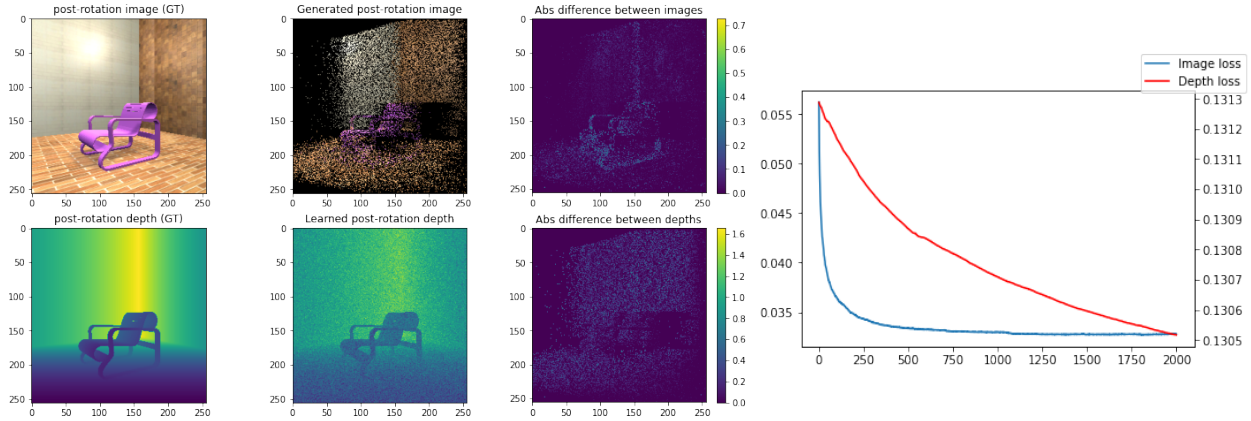

Fig. 3.14. Similar experiment as in Figure 3.13 but where the depth map is initialized by adding noise to the ground truth depth map.

To validate our conclusions, we run a second test using the same setup but where the depth map is initialized to the ground truth with some added Gaussian noise with standard deviation corresponding to 5% of the scene's bounding box length. As Figure 3.14 shows, this experiment exhibits the same issues that the ground truth depth map is not recovered and that the depth map is barely affected by the gradient signal (decreasing by approximately 0.001 while the image MSE loss decreases by over 0.02).

3.5. Discussion and Conclusion

In conclusion, we find that, while promising, our approach does not achieve our two main goals. First, for the task of unsupervised monocular depth estimation, we find that our model fails to generate realistic depth maps and instead produces depth maps that are best suited for the generator to maximize the training objective. Indeed, the produced depth maps seem to minimize the amount of perceived rotation when rendering the surfel point cloud from a rotated point of view. This, in turn, means that our model generates images which do not appear to be from the correct input point of view and instead appear to be from a viewpoint close to the canonical pose.

Exploring the dynamics of the gradients through our differentiable rendering layer, we find that the magnitude of the training signal to the depth maps is too small for our model to learn good depth estimation. This, coupled with the fact that differentiable renderers are notorious for creating non-smooth optimization landscapes with many local minima, leads us to conclude that using surfels as an intermediate representation is not suitable for a HoloGAN-like architecture.

Nevertheless, the recent successes of approaches such as RGBD-GAN and SynSin seem to indicate that there is some value in using point cloud representations as intermediate representations in different contexts. It also indicates that the gradient values coming from the differentiable renderer can contain some meaningful training signal. We thus believe that our approach may find use in slightly different architectures, for example if trained in an autoencoder setup with a dataset of labelled image pairs of scenes similarly to SynSin. Our approach could also be used in a semi-supervised setting or in a transfer learning scenario where only small refinements to the depths would need to be made using the pipeline containing our differentiable surfels renderer.

Chapter 4

A Voronoi-based 3D Representation for Deep Learning

4.1. Introduction

Another sub-problem of unsupervised inverse graphics is the choice of scene representation. As explored in the previous chapter, while implicit representations provide some benefits when it comes to training with neural networks, explicit representations constitute the end goal for inverse graphics because of the wide range of downstream tasks they provide. Nevertheless, it is hard to find an explicit representation that fits all the criteria of unsupervised inverse graphics. Of those, one of the most important is that the representation can be rendered in a differentiable manner as to fit in the unsupervised inverse graphics pipeline highlighted in Figure 1.1. Additionally, the representation needs to be suitable for use in a deep learning setup, meaning that there must exist a way for a neural network to efficiently output this representation. Taking triangle meshes for example, outputting a graph from a neural network is non trivial and requires either a graph neural network architecture which typically performs worse than a regular neural network or a regular MLP but with a fixed-topology mesh where the number of nodes and their connections are pre-determined.

We propose a novel 3D representation that combines the benefits of the most popular representation types which makes it an ideal candidate for use in the inverse graphics pipeline. This representation is based on Voronoi diagrams, is trivial to output from neural networks, is explicit and can be used to model both surface and volumetric quantities. We perform a methodical analysis of the performance of our representation by first evaluating it in the supervised setting using the popular ShapeNet dataset. This allows us to compare our work with the many other approaches recently proposed in the field.

We find that while our approach does not always outperform competing approaches, its other numerous advantages over other representations make it a worthwhile consideration for many unsupervised inverse graphics sub-tasks. We present here performance results of

our approach in the simpler supervised 2D setting, followed by an extensive comparison to prior work in the supervised 3D setting, before finishing by presenting early results in the more challenging weakly supervised 3D setting.

In this chapter, we first analyse existing explicit 3D representations and compare their pros and cons in the preliminaries Section 4.2. We then present our method in Section 4.3 followed by our results in Section 4.4. We conclude by summarizing and discussing our results in Section 4.5.

4.2. Preliminaries

While there exist many different types of 3D representations, no representation is free of disadvantages. In this section, we highlight the most common 3D representations used in deep learning and identify their flaws. This, in turn, positions our novel 3D representation in the context of previous work as our approach is crafted specifically to address these limitations.

The most popular 3D representation type in computer graphics is polygonal meshes. These therefore have the advantage of being very widely-supported by common computer graphics software. Additionally, this representation provides explicit surfaces and can model any topology at arbitrary resolution by increase the number of vertices of the mesh. In the context of deep learning, however, this representation is hard to obtain as an output of a neural network due to its graph nature. For this reason, inverse graphics work such as SoftRas [**34**] [**35**] that use this representation typically resort to using a neural network that outputs a deformation vector for each vertex and applying it to a template mesh. This means that the topology of the mesh is fixed (it corresponds to the template mesh), that the genus of the learned shape can thus not be changed and that the mesh has limited resolution.

Point clouds, on the other hand, are easy to use in a neural network architecture and allow the network to output shapes of arbitrary topology. Nevertheless, the network usually has a fixed-size output, meaning that the number of points in the cloud needs to be pre-determined and cannot vary between outputs. Additionally, this method can become computationally intensive when generating a large number of points. Finally, it can be difficult to extract an explicit surface out of this representation as the point cloud is not always dense enough to recover small details. This representation is used in work such as PointNet and PointFlow [**53**] [**55**] [**69**].

The representation that is easiest to use in a deep learning setting is voxel grids as it is the direct equivalent of a 2D pixel grid in 3D. It can thus be easily generated/processed by a 3D convolutional neural network as is shown in VoxNet [**39**] and work by Qi et al. [**54**]. This representation also has the added benefit of modeling volumetric data while making it very easy to extract from it an explicit surface. Nevertheless, voxel grids are typically severely

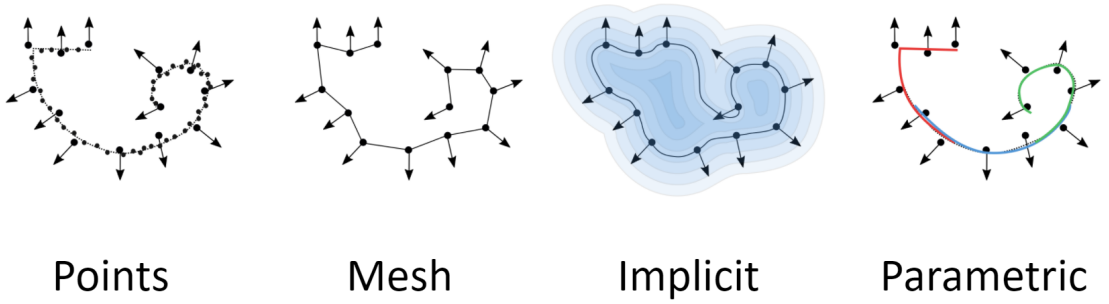

Fig. 4.1. Comparison of popular shape representations in 2D. Here, the large black points and arrows constitute a target shape represented by a point cloud with normals.

computationally limited as their memory and processing requirements grow cubically in three dimensions. In practice, approaches using this type of representation therefore keep the grid resolution low, thus losing out on fine geometrical details.

On the other hand of the 3D representation spectrum, implicit representations model surfaces as zero level sets of neural networks. This provides them many benefits, namely the ability to model shapes of any topology and at infinitely fine resolution (provided the neural network has enough capacity) all while having a relatively small memory footprint. Since this representation is itself a neural network that takes as input a 3D coordinate, it is trivial to use it in a deep autoencoder setup by conditioning this network on a latent code representing the shape, as is done in OccNet [**41**]. Implicit representations have, as their name suggests, the disadvantage that they do not model surfaces explicitly. Surface extraction can be performed using techniques such as marching cubes [**37**], however this algorithm is computationally expensive, not differentiable and can result in a loss of fine details if run at too coarse of a resolution. A sub-category of implicit representations is parametric representations. In this representation, many neural networks are usually used to represent parametric surfaces that take as input a 2D coordinate in the range $[0, 1]$ and outputs a 3D coordinate. By taking the union of these surfaces one can obtain a full 3D model with arbitrary topology. Nevertheless, this type of representation often results in surfaces that are not coherent and that intersect with each other. In addition, it is not trivial to produce a rendering of this representation as there is no reverse mapping from a 3D point on the surface to the corresponding 2D sample point. This representation is used in work such as AtlasNet [**18**] [**12**].

Figure 4.1 shows a comparison of these representations in 2D using a target shape represented by a point cloud with normals.

Out of these, representations that model volumetric quantities also provide an additional advantage. Indeed, in the context of deep unsupervised inverse graphics, volumetric representations are much easier to learn than surface representations. This is because, in an iterative gradient-based optimization scheme, it is much easier to achieve a target shape by

making volume appear and disappear than by moving surfaces towards their target, especially when those surfaces have a fixed topology. This is exemplified in many recent work that outperform surface-based techniques [**41**] [**42**] [**11**].

Inspired by these various types of representations, we explore a new representation based on Voronoi diagrams that addresses a lot of the issues highlighted above. Our representation was specifically designed to model *both* volumetric quantities and explicit surfaces. In addition, it also combines the benefits of point cloud representations as the Voronoi sites can be seen as a point cloud. It has small memory requirements, is easy to output from a neural network, can represent any topology, is simple to use and can be rendered differentiably as we will show in the following section.

4.3. Method

In this section we describe our novel representation, how it can be rendered differentiably and how it can be used in a deep learning architecture both with and without 3D supervision.

4.3.1. The Representation

Using a set of points in space called *seeds*, a Voronoi diagram partitions space into cells in which the closest seed is shared. Mathematically, a cell C_i corresponding to seed $\mathbf{p}_i \in P$ is given by:

$$C_i = \left\{ \mathbf{x} \in \mathbb{R}^3 \mid ||\mathbf{p}_i - \mathbf{x}||_2 < ||\mathbf{p}_j - \mathbf{x}||_2 \text{ for all } i \neq j \right\} \tag{4.3.1}$$

where P is the set of Voronoi seeds. Note that Voronoi cells form convex polytopes.

This partition of plane, while well studied in the fields of mathematics and computational geometry, cannot be directly used to model 3D shapes.

Our 3D representation consists of a Voronoi diagram where each Voronoi cell is given an occupancy value. For scenes without participating media, the occupancy values are binary and object surfaces correspond to the boundaries between Voronoi cells of opposing occupancies. Since a Voronoi diagram is fully defined by a set of *seed* points $\mathbf{p} \in P$, our representation can be stored in a $|P| \times 4$ matrix, where the number of columns corresponds to the dimensionality of the space (3) plus a dimension for the occupancy value. The occupancy can be attached directly to the Voronoi seed as in a Voronoi diagram each seed maps to exactly one Voronoi cell. An example of our representation is illustrated in Figure 4.2 in the easier to visualize 2D setting. To the best of our knowledge we are the first to propose the use of Voronoi diagrams for occupancy function representation.

An explicit surface can be easily obtained from a set of Voronoi seeds by computing the Voronoi tesselation with fast algorithms such as Fortune's algorithm [**15**]. By extracting only faces which separate a cell with occupancy 0 from a cell with occupancy 1 we obtain an

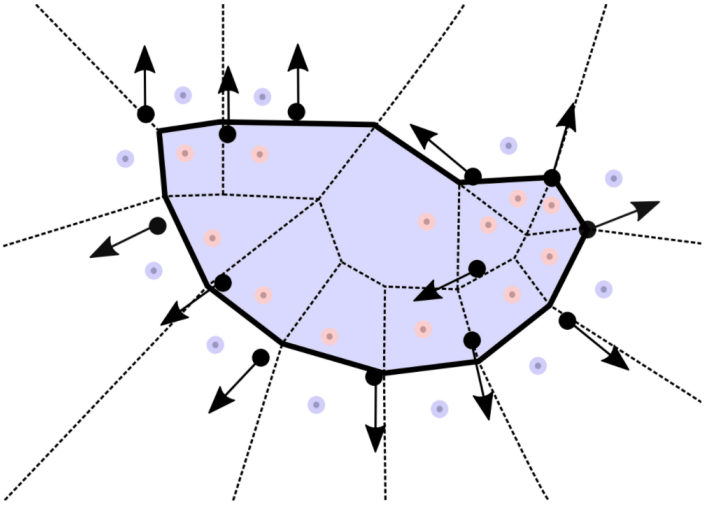

Fig. 4.2. Illustration of our Voronoi representation in 2D. Here, the orange and purple points are the Voronoi seeds which form the Voronoi diagram marked with dotted and solid lines. The resulting surface is illustrated with solid lines and corresponds to the edges of the Voronoi diagram that lie between an orange seed (with occupancy $w(\mathbf{p}) = 1$) and a purple seed (with occupancy $w(\mathbf{p}) = 0$). The black dots and arrows illustrate target point samples and their normal.

explicit surface for the represented shape. This surface can easily be extracted differentiably by computing the equation of the plane forming each face as a function of the two Voronoi seeds it separates (perpendicularly).

This representation can be stored as a point cloud and thus shares the benefit that it is easy to output from a neural network. Additionally, it models an occupancy function similarly to implicit representations while also providing an explicit surface that can trivially be (differentiably) converted to a triangle mesh for downstream applications. This representation is therefore guaranteed to generate watertight meshes. These advantages, combined with the fact that this representation can model shapes or scenes of arbitrary topologies and the fact that it is extremely compact makes it a great fit for 3D deep learning applications and for unsupervised inverse graphics in particular.

4.3.2. Differentiable Rendering

To obtain the occupancy value o at a given point in space $\mathbf{x}$, one simply needs to retrieve the occupancy value stored for the closest Voronoi seed:

$$o(\mathbf{x}) = w\left(\arg\min_{\mathbf{p}\in P} ||\mathbf{x} - \mathbf{p}||_2^2\right) \tag{4.3.2}$$

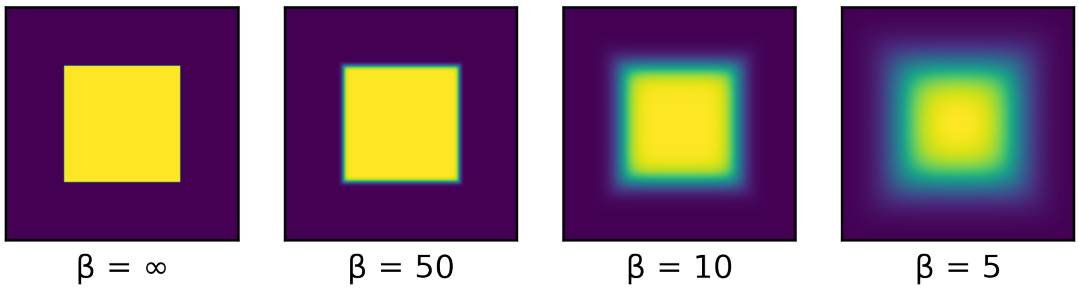

Fig. 4.3. Comparison of the effect of varying the softness parameter β on the rendering of the Voronoi representation. A larger value of β results in an image with sharper edges. The special case of $\beta = \infty$ corresponds to the traditional non-differentiable Voronoi formulation.

For ease of notation, we represent the occupancy value associated with each Voronoi cell as a function $w()$ which takes as input a Voronoi seed as an index. We compute the squared distance between $\mathbf{x}$ and $\mathbf{p}$ as it requires one fewer square root computation.

Note, however, that the process in Equation 4.3.2 is not differentiable with respect to the Voronoi seeds' positions $\mathbf{p}$. To remedy this, we propose a *soft Voronoi* formulation which replaces the non differentiable min operation with a differentiable Softmin:

$$o_{\text{soft}}(\mathbf{x}) = \sum_{\mathbf{p} \in P} w(\mathbf{p}) \frac{e^{-\beta ||\mathbf{x}-\mathbf{p}||_2^2}}{\sum_{\mathbf{p}_2 \in P} e^{-\beta ||\mathbf{x}-\mathbf{p}_2||_2^2}} \tag{4.3.3}$$

where β is a hyperparameter used to control the sharpness of the Voronoi cell edges. The effect of this parameter is shown in a simple 2D example in Figure 4.3.

This equation makes it possible to use our representation in an optimization process where we are given ground truth 3D occupancy values. Nevertheless, we are also interested in unsupervised learning where this representation needs to be rendered as a 2D image. For this we use the volume rendering Equation 2.1.5 but since we only model occupancy values we only render a silhouette image which corresponds to the *transmission* term T_r:

$$T_r(\mathbf{x}_0, \mathbf{x}_f) = \exp\left(- \int_{\mathbf{x}_0}^{\mathbf{x}_f} o_{\text{soft}}(\mathbf{x}) \, \mathrm{d}\, \mathbf{x} \right) \tag{4.3.4}$$

where $\mathbf{x}_0$ and $\mathbf{x}_f$ correspond to the start and end points along a given camera ray, respectively. In practice, Equation 4.3.4 is evaluated numerically by taking N uniformly distributed samples along each ray between points $\mathbf{x}_0$ and $\mathbf{x}_f$.

Evaluating the distance between each queried point $\mathbf{x}$ and each Voronoi seed $\mathbf{p}$ can quickly generate a very large computation graph for automatic differentiation. Taking into account that only very few Voronoi seeds have a sizeable impact on the computed occupancy value at any given point $\mathbf{x}$, in practice we only compute the Softmin operation using the nearest K neighbour seeds of $\mathbf{x}$.

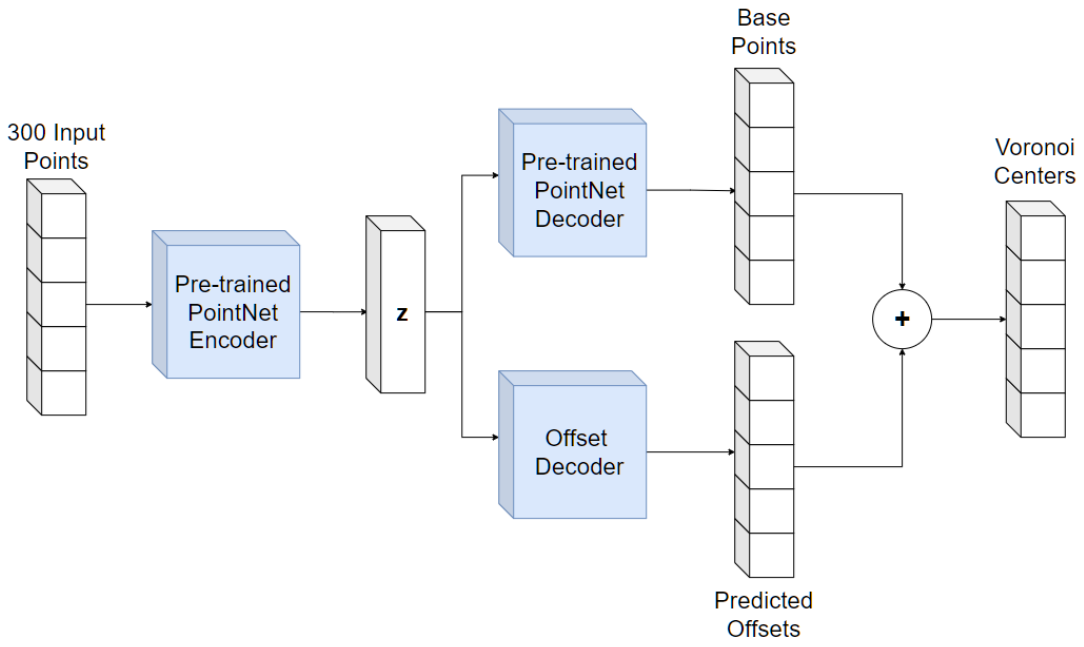

Fig. 4.4. Diagram of our proposed architecture for learning an autoencoder model on the ShapeNet dataset. We first pre-train a PointNet encoder and decoder before adding an offset decoder and training with our full suite of losses.

4.3.3. Architecture

We apply this representation to a handful of applications involving deep learning. First, we demonstrate the effectiveness of our representation in a 2D setting by learning an autoencoding model on a dataset of images of handwritten digits. For this experiment we create a VAE model consisting of a convolutional encoder and a fully connected decoder which outputs a set of Voronoi seeds.

Second, we test our approach against competing methods on the task of 3D shape reconstruction from sparse point clouds. In this context a sparse point cloud of a shape is given as input and the goal is to output the complete reconstructed 3D model. Direct 3D supervision is given in the form of ground truth occupancy samples in space. To process the input point cloud, we employ a PointNet architecture [**55**] identical to that of OccNet against which we compare. We find that we get better results by first pre-training this PointNet autoencoder to output approximately uniformly distributed surface points using only the Chamfer distance [**5**] as training signal. Following this step we train an additional *offset decoder* which consists of an MLP that outputs two displacement vectors for each output of the pre-trained PointNet decoder. For each output of the PointNet decoder, one displacement vector is used to compute the position of a Voronoi seed with occupancy 0 while the other is used for a Voronoi seed with occupancy 1. This encourages the Voronoi seeds to lie close to the object's

surface, therefore maximizing the representation capacity of the Voronoi diagram. This architecture is detailed in Figure 4.4. We train this model using a suite of losses described in the following section.

Third, we provide early results of our approach used in unsupervised setting by evaluating it on a dataset of multiple images of a chair taken from many viewpoints (provided as labels) with the goal of reconstructing a 3D model of the chair. For this experiment we directly optimize a set of randomly initialized Voronoi seed positions and opacities and do not make use of any neural network.

As illustrated in the difference between the second and third settings described above, the occupancy values associated with each learned Voronoi seed can either also be learned or can be fixed to match a pre-determined split. We find that in settings where we can easily get Voronoi seeds to lie close to the object's surface we get better results when using a fixed 50-50 split between cells of occupancy 0 and those of occupancy 1, especially in cases like in our second experiment setting where we can generate a pair of opposing Voronoi seeds from a surface point. In an unsupervised setting such as in our third application however, we find that we get better results when also learning the Voronoi cell occupancies.

4.3.4. Losses

For the first and third experiment settings described in the previous section, we use simple Chamfer and $L1$ losses, respectively. Nevertheless, in the second setting we find that we get our best results by using a suite of losses simultaneously optimizing the positions of Voronoi seeds, the resulting occupancy function and the resulting extracted surface. This is only possible because some 3D supervision data is given in this setting. Using all three of these losses also demonstrates the power of our approach as it is the only one which can learn from point, surface and volume signals all at once, thus combining the benefits of all three types of representations.

The point-based loss is the combination of a Chamfer distance and a *level-set loss*:

$$\mathcal{L}_{\text{point}} = \text{cham}(P, \hat{S}) + \frac{1}{|P|} \sum_{\mathbf{p} \in P} |\text{sdf}(\mathbf{p}) + \alpha(2w(\mathbf{p}) - 1)| \tag{4.3.5}$$

where P is the set of Voronoi seeds, $\hat{S}$ is the set of ground truth surface points, α is a scalar that determines the "height" of the level-set and sdf() is the ground truth signed distance function. The Chamfer distance [**5**] between two point sets A and B is defined as follows:

$$\text{cham}(A, B) = \frac{1}{|A|} \sum_{x_A \in A} \min_{x_B \in B} ||x_A - x_B||_2 + \frac{1}{|B|} \sum_{x_B \in B} \min_{x_A \in A} ||x_A - x_B||_2 \tag{4.3.6}$$

The Chamfer loss component of Equation 4.3.5 is the same as what is being used to pre-train the PointNet autoencoder. After that pre-training step we add the second term of this

equation which corresponds to a *level-set loss*. This loss pushes the Voronoi seeds to lie on the $\pm\alpha$ level-set, with the seeds with occupancy 1 lying on the $-\alpha$ level-set and the ones with occupancy 0 on the $+\alpha$ level-set. This encourages Voronoi seeds to not only lie on the correct side of the surface boundary but also to stay close to the surface, thereby ensuring that no representation capacity is lost.

The occupancy-based loss is a simple $L1$ loss between the computed occupancy at points in space and the ground truth occupancy:

$$\mathcal{L}_{\text{occupancy}} = \frac{1}{|\hat{O}|} \sum_{\mathbf{x} \in \hat{O}} |o_{\text{soft}}(\mathbf{x}) - \hat{o}(\mathbf{x})| \tag{4.3.7}$$

where $\hat{o}$ is the ground truth occupancy function and $\hat{O}$ is the set of ground truth occupancy samples.

The surface-based loss is a simple Chamfer distance between ground truth surface samples $\hat{S}$ and surface samples S drawn differentiably from the extracted Voronoi diagram surface:

$$\mathcal{L}_{\text{surface}} = \text{cham}(S, \hat{S}) \tag{4.3.8}$$

The final loss is a weighted sum of the three terms using hyperparameters λ_{point}, $\lambda_{\text{occupancy}}$ and λ_{surface}:

$$\mathcal{L} = \lambda_{\text{point}} \mathcal{L}_{\text{point}} + \lambda_{\text{occupancy}} \mathcal{L}_{\text{occupancy}} + \lambda_{\text{surface}} \mathcal{L}_{\text{surface}} \tag{4.3.9}$$

4.3.5. Post-processing

We find that some Voronoi cells of occupancy 1 outputted from the decoder can sometimes take disproportionate dimensions, especially around areas of high curvature on the surface of the object. Indeed, because our Voronoi seeds are trained to be concentrated around the $\pm\alpha$ level-sets if the closest Voronoi seeds with occupancy 0 are slightly too far to the side this can result in a Voronoi cell of occupancy 1 which "explodes" and becomes unbounded or very large. To solve this problem, we employ a simple post-processing scheme that detects such Voronoi cells and flips their occupancy value.

We first compute the Voronoi diagram of the output Voronoi seeds. We then flip the occupancy bit of Voronoi cells with occupancy 1 that have infinite volume. For each cell, we also label each vertex with the number of adjacent Voronoi cells with occupancy 1 it touches. This essentially separates the vertices on the surface which are shared with adjacent cells from those which are unique to the current cell. We use this to compute the maximum distance (per cell) between the vertices with one occupancy 1 neighbour and vertices with more than one of such neighbours. If this distance is above a given cut-off distance threshold t_1, we flip the occupancy bit of the current cell. Finally, we flip the occupancy of cells for

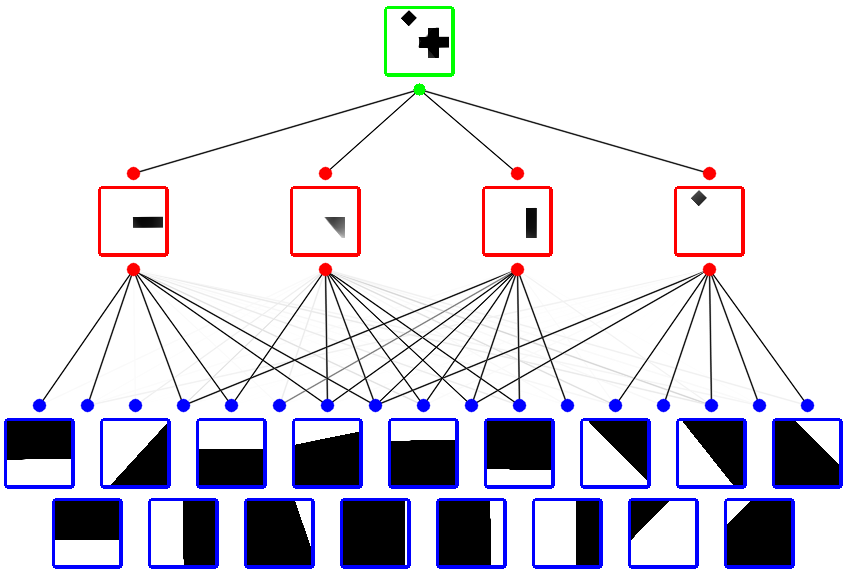

Fig. 4.5. The BSP-Net representation in 2D. The set of planes is represented in the blue squares while the learned convexes are shown in red. The final shape, shown in green, is the union of the convexes. Image from BSP-Net [**9**].

which the ratio of Voronoi faces which form object faces is above a certain threshold t_2. Note that we evaluate this for all Voronoi cells *before* any occupancy flipping is applied.

4.3.6. Concurrent Work

Concurrently to our work, two other representations have been proposed that share the property of being simultaneously implicit and explicit. Neither of these methods, however, can leverage point-based, surface-based and volume-based supervision simultaneously like our Voronoi representation.

Deng et al. propose **CvxNet** [**11**], a 3D representation and deep learning architecture based on the union of a collection of convex shapes. In their proposed approach, a neural network outputs a set of convexes, each of which defined by the intersection of a set of half spaces. The occupancy function defined by these convexes is evaluated differentiably at a given point by computing the perpendicular distance from this point to each plane. Each distance is then passed through a sigmoid function before going through a Softmin operation to differentiably model the half space intersection computation. While this approach can also model shapes of arbitrary shape and topology, it suffers from self-intersection problems. Indeed, each learned convex can overlap with adjacent convexes leading to problems when extracting an explicit surface for downstream tasks. In contrast, our Voronoi-based approach does not suffer from this problem as our convex cells are guaranteed to be non-overlapping.

Chen et al. propose in **BSP-Net** [**9**] a novel representation based on binary space-partitioning trees. Their representation consists of the union of a set of convex primitives, where each of those convexes are defined by the intersection of a set of planes. In a deep learning setting, they design a mechanism to learn this representation by making the shape formation process differentiable. A set of space-partitioning planes is learned and each convex is defined by a linear combination of those planes. The weights of the linear combination are learned and are replaced at test time with binary weights to generate objects with sharp and well-defined boundaries. This process is highlighted in Figure 4.5.

4.4. Results

We evaluate our representation on three different settings. We first validate our approach in a 2D setting. We then compare our approach with recent work on a supervised 3D reconstruction task. Finally, we show how our representation can be used in an unsupervised 3D setting. We implement our approach and all of our models in PyTorch [**49**]. The nearest-neighbour queries are performed using FAISS [**24**] for fast inference speed on the GPU.

4.4.1. Supervised 2D Experiments

We first train a VAE model to output a 2D version of our Voronoi representation describing hand-written digits from the MNIST dataset. We refer to this experiment as "supervised" since the gray-scale pixel values from the MNIST images are a form of ground truth occupancy function. We use a fully convolutional encoder and an MLP decoder and train using the Adam optimizer with a learning rate of 10^{-4}. We use the $L1$ loss as training signal to compare our rendered occupancy image to the ground truth digit image. The decoder outputs the 2D coordinates for 128 Voronoi sites and the Voronoi cell occupancies are fixed with a 50-50 split. We additionally compute a ground truth signed distance function discretized on a grid by calculating the distance to the closest pixel of opposing occupancy. We use this SDF in an additional training signal consisting of the level-set loss described in Equation 4.3.5 with an α parameter corresponding to 1% of the image width. We use $\beta = 10^4$ for evaluating our soft occupancy function.

We compare our approach to a 2D implementation of OccNet [**41**] with a varying number of parameters. Figure 4.6 shows a qualitative overview of our results and comparison to OccNet models. Table 4.1 shows the quantitative results of this experiment. We find that our approach performs similarly to OccNet with two orders of magnitude fewer parameters for the representation and that our approach significantly outperforms OccNet when comparing with equal number of parameters.

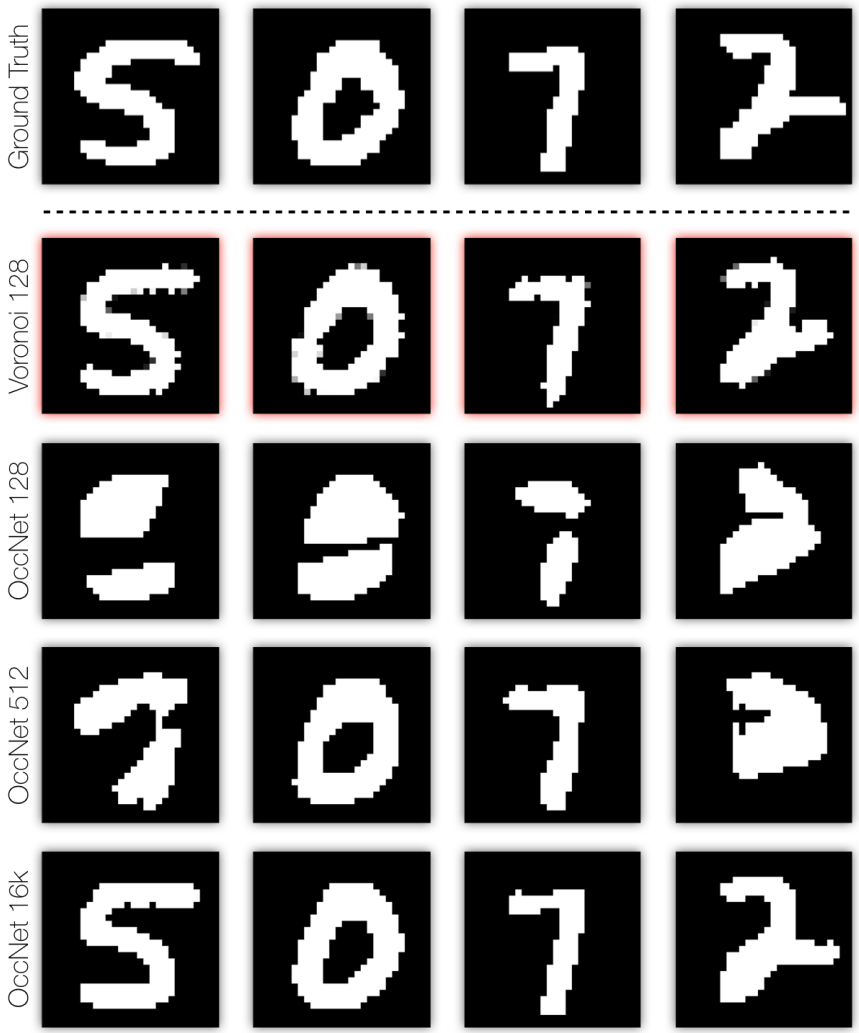

Fig. 4.6. Qualitative results of our autoencoding approach on the MNIST dataset. Compared to OccNet, we are able to achieve comparable accuracy with representations containing two orders of magnitude less parameters (128 vs 16k).

Method	Mean	Std	Med
OccNet 128	83.80	28.21	85.69
OccNet 512	76.17	28.21	75.42
OccNet 16k	52.66	14.33	53.04
Voronoi 128	58.00	17.02	58.29

Table 4.1. Quantitative results of our autoencoding approach on the MNIST dataset, compared to OccNet. Here the mean, standard deviation (*Std*) and median (*Med*) are computed by summing the absolute differences between the generated and target pixels.

4.4.2. Supervised 3D Experiments

We then test our approach on a supervised 3D setting using the ShapeNet airplane dataset [**7**]. This allows us to directly compare our results with OccNet and AtlasNet, two competing approaches that require 3D supervision. The task explored here is 3D shape reconstruction from a sparse point cloud of 300 surface points. We use surface point samples, an SDF function and occupancy volume samples as ground truth. As described in Section 4.3.3, we use the same PointNet architecture as OccNet to process the point cloud input. We train two versions of our autoencoder model. One that outputs 1024 Voronoi seeds and one that outputs 10000 seeds. We train our network in two stages. First, the PointNet autoencoder is pre-trained using the Chamfer distance metric using each iteration a number of samples from the ground truth set of surface points $\hat{S}$ equal to the number of generated Voronoi seeds. We train until convergence using the Adam optimizer with a learning rate of 10^{-4} and with a batch size of 32. Second, we train our full architecture using the same optimizer and learning rate. We use a level-set distance parameter α of 0.01, a blurring parameter β of 10^4, and hyperparameter values $\lambda_{\text{point}} = 0.5$, $\lambda_{\text{occupancy}} = 0.01$ and $\lambda_{\text{surface}} = 0.001$. Each iteration we evaluate the occupancy loss $\mathcal{L}_{\text{occupancy}}$ using 10000 ground truth occupancy volume samples and the surface loss using 1000 surface samples. We train until convergence which takes around 5 days using a single V100 GPU.

Figure 4.7 shows qualitative results of our approach compared to OccNet. Table 4.2 shows quantitative results compared to both AtlasNet and OccNet. Note that we use the Chamfer distance (a surface metric) to compare our results to AtlasNet and the IoU metric (a volume metric) to compare to OccNet as AtlasNet does not output an occupancy function and it is not easy to obtain surface samples from a trained OccNet model. We evaluate both the Chamfer and IoU losses using 10k randomly sampled points on the surface and volume, respectively. We find a similar conclusion to this experiment as in the supervised 2D setting. Our model performs well even using a small number of seeds which in this case corresponds to four orders of magnitude fewer parameters required to model an object. We find that our best model (using 10k seeds) performs slightly worse than OccNet and that the standard deviation of the IoU metric specifically could be improved. We attribute this to the cell explosion problem for which we designed the post-processing method. Despite the small performance gap between our approach and OccNet, our method provides additional benefits such as an explicit (and differentiable) surface.

Compared to AtlasNet, Table 4.2 shows that our approach again performs slightly worse than AtlasNet with the exception of the median Chamfer distance metric for our 10k seeds model. This indicates that if we can address the instability issue or design a better post-processing method, we could bridge the gap between our approach and AtlasNet. This is an interesting research direction that we leave as future work.

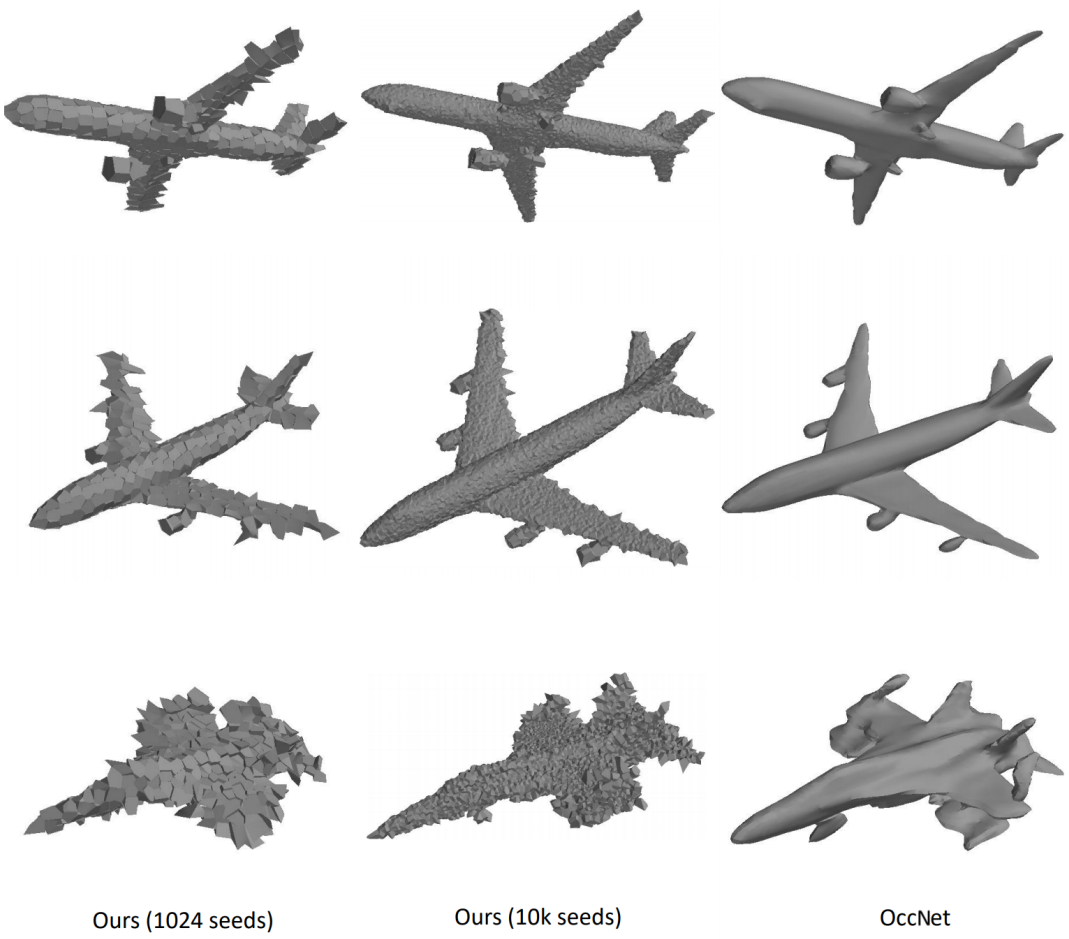

Fig. 4.7. Results of our autoencoding approach on the ShapeNet airplanes dataset. We show the results of two models, one trained with 1024 seeds and one with 10k seeds. Our model is able to represent the shape well even using a low number of seeds.

	Cham mean	**Cham std**	**Cham med**	**IoU mean**	**IoU std**	**IoU med**
AtlasNet	0.00081	0.00060	0.00062	-	-	-
Ours (1024)	0.00487	0.01193	0.00190	0.69851	0.20030	0.73198
Ours (10k)	0.00142	0.00282	0.00048	0.70402	0.21565	0.74232
OccNet	-	-	-	0.77297	0.12075	0.79915

Table 4.2. Quantitative results of our autoencoding approach on the ShapeNet airplanes dataset. We compare with AtlasNet, a parametric representation approach, using the Chamfer distance with 2500 sampled surface points. We report the mean, standard deviation (std) and median (med) for all metrics. We compare with OccNet, an implicit representation approach, using the IoU loss using 100k uniform volume samples. We observe that our model with 10k Voronoi seeds performs slightly worse than both competing approaches while providing other advantages. Nevertheless, we note that our approach outperforms AtlasNet on the median Chamfer distance metric.

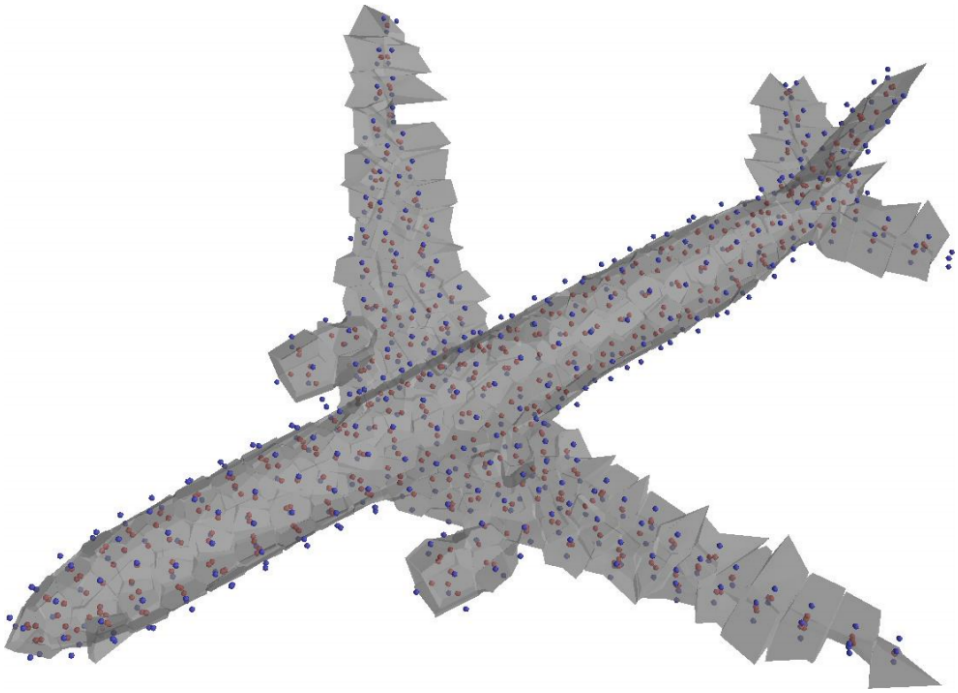

Fig. 4.8. Visualization of output Voronoi seeds of a model trained on ShapeNet airplanes. The blue points represent the Voronoi seed with occupancy 0 while the red points represent those with occupancy 1. Notice how the Voronoi seeds lie on two distinct level-sets of the model.

We show an example learned airplane and its corresponding Voronoi sites in Figure 4.8. The Voronoi seeds appear evenly distributed and lying on two distinct level sets of the shape. This figure also shows that our approach could benefit from having more densely distributed Voronoi seeds around thin structures and areas of high curvatures such as the wings and tail of the plane.

We additionally run a simple direct optimization experiment to compare the capacity of our representation to OccNet's. We directly optimize our Voronoi representation using varying size to match a 3D sphere using only the occupancy loss. We do the same with an OccNet neural network. Figure 4.9 the results of this experiment and that our approach performs a lot better than OccNet at low number of parameters.

4.4.3. Unsupervised 3D Experiments

Finally, we test our representation on an unsupervised 3D setting. We note, however, that our results are on simple experiments and that further experimentation will be necessary to thoroughly compare our approach to that of other 3D unsupervised methods.

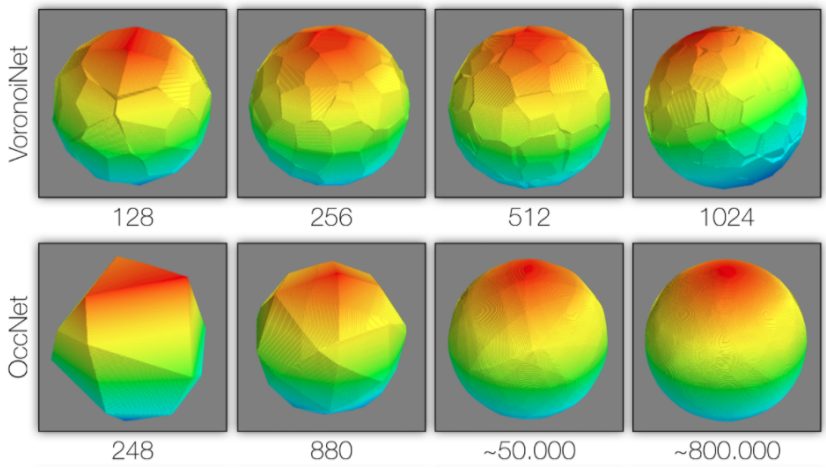

Fig. 4.9. Comparison of the representation capacity of our approach compared to OccNet in a 3D setting. In this experiment, we overfit both our representation and an OccNet model to a 3D unit sphere and show that our approach represents the model more accurately at low parameter counts.

We directly optimize a set of Voronoi seeds $\mathbf{p}$ and their occupancies $w(\mathbf{p})$ by rendering this representation to a 2D silhouette image from various viewpoints and comparing those against a dataset of ground truth silhouette renderings of a 3D chair model. We optimize this using SGD with a momentum of 0.9 and a learning rate of 0.1 for the seed positions $\mathbf{p}$ and 20 for the occupancies $w(\mathbf{p})$. We compare the rendered silhouette with the ground truth using the MSE loss. We use 1000 Voronoi seeds and constrain their positions to be within the bounding box of the scene by passing them through a scaled sigmoid function. We produce a silhouette rendering by computing the total transmission along each camera ray by approximating Equation 4.3.4 numerically using 300 uniformly distributed samples per ray. Additionally, since in this context $w(\mathbf{p})$ corresponds to the *extinction coefficient* of each cell rather than their occupancies, we pass these values through an exponential function prior to using them in the rendering equation. We use $\beta = 4 \times 10^2$ for differentiable rendering.

The results of this experiment are shown in Figure 4.10. Below the ground truth silhouette images of the chair model are shown rasterized renderings of the mesh model extracted from the Voronoi representation at the end of training. We find that our model is able to reconstruct a shape that closely matches the ground truth silhouettes. Nevertheless, without any additional smoothness prior or regularization loss our representation tends to generate blocky surfaces. Adding such a regularizer is a potential area of improvement for future work.

Fig. 4.10. 360 degree visualization of a learned chair represented using our Voronoi-based representation. The top row shows the ground truth silhouette while the bottom row shows the final learned 3D model for the same poses.

4.5. Discussion and Conclusion

Overall, our novel Voronoi-based representation provides many benefits over other types of representations. Namely, it combines the benefits of point-based, surface-based and volume-based representations. It can model shapes and scenes of arbitrary topology and can be rendered differentiably, on top of simultaneously representing shapes implicitly and explicitly. All these advantages make it particularly well suited for unsupervised inverse graphics applications. Despite the slight performance gap when compared to OccNet (a competing implicit representation approach), we believe our Voronoi-based representation has a lot of potential for use in deep inverse graphics pipeline, especially since it makes it possible to obtain an explicit differentiably for downstream tasks such as physics and lighting simulations, something that OccNet does not support.

In an unsupervised setting, we demonstrated that our approach performs well on a simple direct optimization experiment, but much still needs to be done to fully assess the performance of our methods compared to other unsupervised approaches such as SoftRas [**34**] and DIB-R [**8**]. In particular, we would like to see how this representation compares on the same unsupervised ShapeNet object silhouettes as SoftRas. We also think that there is a lot of potential in applying additional regularizers to this setting to encourage smoother output surfaces.

As highlighted in the previous section, the biggest challenge with our approach is training instability. Because small movements in Voronoi seeds can lead to very large changes in cell surfaces, we found that our approach has more trouble training smoothly and generating shapes with thin edges. This results in higher variance in our generated results which brings down our overall accuracy. While the use of a hand crafted post-processing step mitigates

this problem, it is far from perfect and we would like to see if this cannot be improved through other means.

Chapter 5

Conclusion

In this work, we explored the use of two different 3D representations in the context of deep unsupervised inverse graphics: one novel use of an existing representation named *surfels* and one novel representation based on Voronoi diagrams. Before explaining our approaches and results in depth, we first explained the problem and goals in detail along with relevant background information. Additionally, we provided an overview of the current state of the art in the field and identified issues with current approaches.

The first method we presented combined the work of HoloGAN [**44**] and Pix2Shape [**56**] in an attempt to improve upon the state of the art on the task of novel-view synthesis, a sub-task of inverse graphics. In this project we reproduced the HoloGAN generative pipeline while replacing the implicit intermediary 3D voxel representation with an explicit surfel representation. We justified our approach by explaining that using an explicit representation would guarantee view consistency, something that we found was an issue in HoloGAN. We showed experiments qualitatively comparing our approach to that of HoloGAN and reached the conclusion that our approach was unable to compete with other work on this setting. To back our claim, we ran an extensive set of experiments to understand the impact of our hyperparameters as well as potential causes for the poor performance we observed. We concluded by stating that although our differentiable surfel renderer experienced training issues, as significant as they were in the unsupervised setting we were targeting, it may still hold value as it has shown to be successful in two concurrent work operating in the weakly supervised setting [**47**] [**67**].

In the second method we presented, we introduced a novel 3D representation based on Voronoi diagrams with an additional per-cell occupancy bit. We showed that this representation has many advantages compared to prior approaches, including that it models surfaces both implicitly and explicitly simultaneously, that it is very compact in terms of memory, that it can model shapes of arbitrary topology and that it can be rendered differentiably. We showed results of this approach on a supervised 2D setting, supervised 3D setting and

unsupervised 3D setting. Although our approach did not perform as well as competing approaches on the supervised task of 3D shape reconstruction from sparse point clouds, our approach is more amenable to downstream tasks as it can model surfaces explicitly. We concluded by identifying interesting areas for future work, notably in the unsupervised setting where more extensive experiments have yet to be performed to truly assess the performance of our method compared to prior work.

The task of unsupervised inverse graphics holds the promise of a future where machine learning models can better interpret 2D images by fully understanding the 3D content and the image formation process behind them. In the short term, it could provide significant benefits to content creation pipelines by making it much faster to capture real-world elements for use in digital productions. In the long term, this technology could be used to improve object detection and image segmentation for applications such as self-driving cars where accuracy is critical for the safety of the system.

Every year, this research topic progresses at a faster and faster pace, which makes us very excited for the future of inverse graphics. The choice of 3D representation is still an area of critical importance within this rapidly evolving topic and we hope that our work can provide insight and prove to be useful to the future of deep unsupervised inverse graphics.

5.1. Future Work

Over the past few months, the field of inverse graphics has shifted more and more towards implicit representations. NeRF by Mildenhall et al. [**42**] was a catalyst in this transition as this work showed impressive results that, when rendered, were barely distinguishable from real photos. NeRF was one of the first few work to propose an implicit representation that modeled not only shape, but also appearance by using an MLP to learn the mapping from 3D coordinates to an occupancy/opacity and emitted radiance. Using this representation, they were able to realistically reconstruct a 3D scene from a small number (≤ 100) of posed images by rendering images through volume rendering with ray marching and training the MLP to overfit the scene.

Prior to NeRF, implicit representations typically suffered from blurriness issues, similarly to VAE models on 2D images. The authors proposed a *positional embedding* layer to increase the frequency content of the input coordinate which resulted in a great improvement to the sharpness of the learned representation. This work was followed by similar ideas, notably SIREN networks [**62**] and a follow-up to NeRF exploring Fourier features embedding [**64**].

These methods all approach the task of single scene reconstruction from multi-view images by directly optimizing the representation for a single scene. For unsupervised inverse graphics from a single 2D image, however, using an autoencoding approach trained on a dataset of multiple scenes is necessary. This was first done in OccNet [**41**] and DeepSDF [**48**]

by conditioning the MLP implicit representation on a latent code representing the scene. More recently, MetaSDF [**61**] explored how meta-learning can be used in this context to output the weights of the MLP based on a latent code of the scene. These methods all operate using 3D supervision or with silhouette images as they do not model appearance as is done in NeRF. GRAF [**60**] is the first work to extend the NeRF model to a generative model. Although it lacks an inference mechanism, it is a promising first step towards using the full power of implicit representations in the context of unsupervised inverse graphics.

Implicit representations are really promising because of their simplicity but also because of their training dynamics. Indeed, training neural networks as general function approximators is now a common task since the advent of deep learning and their effectiveness has been shown through both theory and applications. The next challenge for neural implicit representations is to improve the (volume) rendering speed as this ray-marching step is currently a major bottleneck for training, but also at test time where rendering times are still three orders of magnitude too slow for real-time applications. Additionally, more research will be necessary to find ways to properly disentangle object identity from pose since the HoloGAN approach is not applicable to a neural implicit representation.

Although it is difficult to obtain an explicit triangle mesh from an implicit representations, it is possible that implicit representations will become the new standard in the future, thereby removing the need for explicit representations in the first place. This is currently far-fetched, however, as it would require new methods for scene manipulation, animation, rendering, physics simulation, 3D modeling and many more. Nevertheless, this type of representation is currently leading the way for improvements on some specific deep learning applications such as novel-view synthesis and 3D scene understanding in the near future.

References

[1] Kara-Ali Aliev, Artem Sevastopolsky, Maria Kolos, Dmitry Ulyanov et Victor Lempitsky : Neural point-based graphics. *arXiv preprint arXiv:1906.08240*, 2019.

[2] Martin Arjovsky, Soumith Chintala et Léon Bottou : Wasserstein generative adversarial networks. *International Conference on Machine Learning (ICML)*, 2017.

[3] Matan Atzmon et Yaron Lipman : Sal: Sign agnostic learning of shapes from raw data. *arXiv preprint arXiv:1911.10414*, 2019.

[4] Matan Atzmon et Yaron Lipman : Sal++: Sign agnostic learning with derivatives. *arXiv preprint arXiv:2006.05400*, 2020.

[5] Harry G Barrow, Jay M Tenenbaum, Robert C Bolles et Helen C Wolf : Parametric correspondence and chamfer matching: Two new techniques for image matching. Rapport technique, SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER, 1977.

[6] Mojtaba Bemana, Karol Myszkowski, Hans-Peter Seidel et Tobias Ritschel : X-fields: Implicit neural view-, light- and time-image interpolation. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia 2020)*, 39(6), 2020.

[7] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi et Fisher Yu : Shapenet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012*, 2015.

[8] Wenzheng Chen, Huan Ling, Jun Gao, Edward Smith, Jaakko Lehtinen, Alec Jacobson et Sanja Fidler : Learning to predict 3d objects with an interpolation-based differentiable renderer. *In Advances in Neural Information Processing Systems*, pages 9609–9619, 2019.

[9] Zhiqin Chen, Andrea Tagliasacchi et Hao Zhang : Bsp-net: Generating compact meshes via binary space partitioning. *arXiv:1911.06971*, 2019.

[10] Thomas Davies, Derek Nowrouzezahrai et Alec Jacobson : Overfit neural networks as a compact shape representation. *arXiv preprint arXiv:2009.09808*, 2020.

[11] Boyang Deng, Kyle Genova, Soroosh Yazdani, Sofien Bouaziz, Geoffrey Hinton et Andrea Tagliasacchi : Cvxnet: Learnable convex decomposition. *arXiv:1909.05736*, 2019.

[12] Theo Deprelle, Thibault Groueix, Matthew Fisher, Vladimir G. Kim, Bryan C. Russell et Mathieu Aubry : Learning elementary structures for 3d shape generation and matching. *arXiv preprint arXiv:1908.04725*, 2019.

[13] Jeff Donahue, Philipp Krähenbühl et Trevor Darrell : Adversarial feature learning. *arXiv preprint arXiv:1605.09782*, 2016.

[14] Vincent Dumoulin, Ishmael Belghazi, Ben Poole, Olivier Mastropietro, Alex Lamb, Martin Arjovsky et Aaron Courville : Adversarially learned inference. *arXiv preprint arXiv:1606.00704*, 2016.

[15] S Fortune : A sweepline algorithm for voronoi diagrams. *In Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, page 313–322, New York, NY, USA, 1986. Association for Computing Machinery.

[16] Ian Goodfellow, Yoshua Bengio, Aaron Courville et Yoshua Bengio : *Deep learning*, volume 1. MIT Press, 2016.

[17] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville et Yoshua Bengio : Generative adversarial nets. *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[18] Thibault Groueix, Matthew Fisher, Vladimir G Kim, Bryan C Russell et Mathieu Aubry : Atlasnet: A papier-mache approach to learning 3d surface generation. *arXiv:1802.05384*, 2018.

[19] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin et Aaron C Courville : Improved training of wasserstein gans. *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[20] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler et Sepp Hochreiter : Gans trained by a two time-scale update rule converge to a local nash equilibrium. *arXiv preprint arXiv:1706.08500*, 2017.

[21] Berthold K. P. Horn : *Obtaining Shape from Shading Information*, page 123–171. MIT Press, Cambridge, MA, USA, 1989.

[22] Eldar Insafutdinov et Alexey Dosovitskiy : Unsupervised learning of shape and pose with differentiable point clouds. *CoRR*, abs/1810.09381, 2018.

[23] Max Jaderberg, Karen Simonyan, Andrew Zisserman et Koray Kavukcuoglu : Spatial transformer networks. *arXiv preprint arXiv:1506.02025*, 2015.

[24] Jeff Johnson, Matthijs Douze et Hervé Jégou : Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[25] James T. Kajiya : The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, août 1986.

[26] Tero Karras, Timo Aila, Samuli Laine et Jaakko Lehtinen : Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

[27] Tero Karras, Samuli Laine et Timo Aila : A style-based generator architecture for generative adversarial networks. *arXiv preprint arXiv:1812.04948*, 2018.

[28] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl et Adrien Gaidon : Differentiable rendering: A survey. *arXiv preprint arXiv:2006.12057*, 2020.

[29] Hiroharu Kato, Yoshitaka Ushiku et Tatsuya Harada : Neural 3d mesh renderer. *In Proc. of Comp. Vision and Pattern Recognition (CVPR)*, 2018.

[30] Michael Kazhdan, Matthew Bolitho et Hugues Hoppe : Poisson surface reconstruction. *In Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, SGP '06, page 61–70, Goslar, DEU, 2006. Eurographics Association.

[31] Diederik P Kingma et Max Welling : Auto-encoding variational bayes. *arXiv preprint arXiv:undefined*, 2013.

[32] Leif Kobbelt et Mario Botsch : A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004.

[33] Tzu-Mao Li, Miika Aittala, Frédo Durand et Jaakko Lehtinen : Differentiable monte carlo ray tracing through edge sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):222:1–222:11, 2018.

[34] Shichen Liu, Weikai Chen, Tianye Li et Hao Li : Soft rasterizer: Differentiable rendering for unsupervised single-view mesh reconstruction. *CoRR*, abs/1901.05567, 2019.

[35] Shichen Liu, Tianye Li, Weikai Chen et Hao Li : Soft rasterizer: A differentiable renderer for image-based 3d reasoning. *CoRR*, abs/1904.01786, 2019.

[36] Matthew M Loper et Michael J Black : Opendr: An approximate differentiable renderer. *In European Conference on Computer Vision*, pages 154–169. Springer, 2014.

[37] William E Lorensen et Harvey E Cline : Marching cubes: A high resolution 3d surface construction algorithm. *In ACM Trans. on Graphics (Proc. of SIGGRAPH)*, 1987.

[38] Guillaume Loubet, Nicolas Holzschuch et Wenzel Jakob : Reparameterizing discontinuous integrands for differentiable rendering. *ACM Transactions on Graphics*, décembre 2019.

[39] Daniel Maturana et Sebastian Scherer : Voxnet: A 3d convolutional neural network for real-time object recognition. pages 922–928, 09 2015.

[40] Morgan McGuire et Louis Bavoil : Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques (JCGT)*, 2(2):122–141, December 2013.

[41] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin et Andreas Geiger : Occupancy networks: Learning 3d reconstruction in function space. *arXiv:1812.03828*, 2018.

[42] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi et Ren Ng : Nerf: Representing scenes as neural radiance fields for view synthesis. *arXiv preprint arXiv:2003.08934*, 2020.

[43] Mehdi Mirza et Simon Osindero : Conditional generative adversarial nets. *Arxiv*, 2014.

[44] Thu Nguyen-Phuoc, Chuan Li, Lucas Theis, Christian Richardt et Yong-Liang Yang : Hologan: Unsupervised learning of 3d representations from natural images. *arXiv preprint arXiv:1904.01326*, 2019.

[45] Merlin Nimier-David, Sébastien Speierer, Benoît Ruiz et Wenzel Jakob : Radiative backpropagation: an adjoint method for lightning-fast differentiable rendering. *ACM Transactions on Graphics (TOG)*, 39(4):146–1, 2020.

[46] Merlin Nimier-David, Delio Vicini, Tizian Zeltner et Wenzel Jakob : Mitsuba 2: A retargetable forward and inverse renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6), novembre 2019.

[47] Atsuhiro Noguchi et Tatsuya Harada : Rgbd-gan: Unsupervised 3d representation learning from natural image datasets via rgbd image synthesis. *arXiv preprint arXiv:1909.12573*, 2019.

[48] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe et Steven Lovegrove : Deepsdf: Learning continuous signed distance functions for shape representation. *Proc. of Comp. Vision and Pattern Recognition (CVPR)*, 2019.

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai et Soumith Chintala : Pytorch: An imperative style, high-performance deep learning library. *In* H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox et R. Garnett, éditeurs : *Proc. of Neural Information Processing Systems (NeurIPS)*, pages 8024–8035. Curran Associates, Inc., 2019.

[50] Georgios Pavlakos, Luyang Zhu, Xiaowei Zhou et Kostas Daniilidis : Learning to estimate 3d human pose and shape from a single color image. *arXiv preprint arXiv:1805.04092*, 2018.

[51] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar et Markus Gross : Surfels: Surface elements as rendering primitives. *In Annual Conference on Computer Graphics and Interactive Techniques*, 2000.

[52] Matt Pharr, Wenzel Jakob et Greg Humphreys, éditeurs. *Physically Based Rendering (Third Edition)*. Morgan Kaufmann, Boston, third edition édition, 2017.

[53] Charles R Qi, Hao Su, Kaichun Mo et Leonidas J Guibas : Pointnet: Deep learning on point sets for 3d classification and segmentation. *In Proc. of Comp. Vision and Pattern Recognition (CVPR)*, 2017.

[54] Charles R. Qi, Hao Su, Matthias Niessner, Angela Dai, Mengyuan Yan et Leonidas J. Guibas : Volumetric and multi-view cnns for object classification on 3d data. *arXiv preprint arXiv:1604.03265*, 2016.

[55] Charles Ruizhongtai Qi, Li Yi, Hao Su et Leonidas J. Guibas : Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *CoRR*, abs/1706.02413, 2017.

[56] Sai Rajeswar, Fahim Mannan, Florian Golemo, Jérôme Parent-Lévesque, David Vazquez, Derek Nowrouzezahrai et Aaron Courville : Pix2shape: Towards unsupervised learning of 3d scenes from images using a view-based representation. *arXiv preprint arXiv:2003.14166*, 2020.

[57] Olaf Ronneberger, Philipp Fischer et Thomas Brox : U-net: Convolutional networks for biomedical image segmentation. *arXiv preprint arXiv:1505.04597*, 2015.

[58] Shunsuke Saito, Zeng Huang, Ryota Natsume, Shigeo Morishima, Angjoo Kanazawa et Hao Li : Pifu: Pixel-aligned implicit function for high-resolution clothed human digitization. *arXiv:1905.05172*, 2019.

[59] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford et Xi Chen : Improved techniques for training gans. *arXiv preprint arXiv:1606.03498*, 2016.

[60] Katja Schwarz, Yiyi Liao, Michael Niemeyer et Andreas Geiger : Graf: Generative radiance fields for 3d-aware image synthesis. *arXiv preprint arXiv:2007.02442*, 2020.

[61] Vincent Sitzmann, Eric R. Chan, Richard Tucker, Noah Snavely et Gordon Wetzstein : Metasdf: Meta-learning signed distance functions. *arXiv preprint arXiv:2006.09662*, 2020.

[62] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell et Gordon Wetzstein : Implicit neural representations with periodic activation functions. *arXiv preprint arXiv:2006.09661*, 2020.

[63] Vincent Sitzmann, Michael Zollhöfer et Gordon Wetzstein : Scene representation networks: Continuous 3d-structure-aware neural scene representations. *arXiv preprint arXiv:1906.01618*, 2019.

[64] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron et Ren Ng : Fourier features let networks learn high frequency functions in low dimensional domains. *arXiv preprint arXiv:2006.10739*, 2020.

[65] Ayush Tewari, Ohad Fried, Justus Thies, Vincent Sitzmann, Stephen Lombardi, Kalyan Sunkavalli, Ricardo Martin-Brualla, Tomas Simon, Jason Saragih, Matthias Niessner, Rohit Pandey, Sean Fanello, Gordon Wetzstein, Jun-Yan Zhu, Christian Theobalt, Maneesh Agrawala, Eli Shechtman, Dan B Goldman et Michael Zollhöfer : State of the art on neural rendering. *arXiv preprint arXiv:2004.03805*, 2020.

[66] Peng-Shuai Wang, Yang Liu, Yu-Xiao Guo, Chun-Yu Sun et Xin Tong : O-cnn: Octree-based convolutional neural networks for 3d shape analysis. *arXiv preprint arXiv:1712.01537*, 2017.

[67] Olivia Wiles, Georgia Gkioxari, Richard Szeliski et Justin Johnson : Synsin: End-to-end view synthesis from a single image. *arXiv preprint arXiv:1912.08804*, 2019.

[68] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang et Jianxiong Xiao : 3d shapenets: A deep representation for volumetric shapes. *arXiv preprint arXiv:undefined*, 2014.

[69] Guandao Yang, Xun Huang, Zekun Hao, Ming-Yu Liu, Serge Belongie et Bharath Hariharan : Pointflow: 3d point cloud generation with continuous normalizing flows. *arXiv preprint arXiv:1906.12320*, 2019.

[70] Wang Yifan, Felice Serena, Shihao Wu, Cengiz Öztireli et Olga Sorkine-Hornung : Differentiable surface splatting for point-based geometry processing. *arXiv preprint arXiv:1906.04173*, 2019.

[71] Ilker Yildirim, Tejas Kulkarni, Winrich Freiwald et Joshua Tenenbaum : Efficient analysis-by-synthesis in vision: A computational framework, behavioral tests, and comparison with neural representations. 07 2015.

[72] Fisher Yu, Ari Seff, Yinda Zhang, Shuran Song, Thomas Funkhouser et Jianxiong Xiao : Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv preprint arXiv:1506.03365*, 2015.

[73] Cheng Zhang, Bailey Miller, Kai Yan, Ioannis Gkioulekas et Shuang Zhao : Path-space differentiable rendering. *ACM Transactions on Graphics (TOG)*, 39(4):143–1, 2020.

[74] Shuang Zhao, Wenzel Jakob et Tzu-Mao Li : Physics-based differentiable rendering: From theory to implementation. *In ACM SIGGRAPH 2020 Courses*, SIGGRAPH 2020, New York, NY, USA, 2020. Association for Computing Machinery.