UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Research project report title page

Candidate **2494S**

"Efficient colour representation for Gaussian Splatting"

Submitted in partial fulfillment of the requirements for the
Computer Science Tripos, Part III

Total page count: 46

Main chapters (excluding front-matter, references and appendix): 37 pages (pp 7–43)

Main chapters word count: 10732[1]

Declaration

I, **2494S**, being a candidate for Part III of the Computer Science Tripos, hereby declare that this project report and the work described in it are my own work, unaided except as may be specified below, and that the project report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this project report I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my project report to be made available to the students and staff of the University.

Date: 28 May 2024

[1]Taken as the sum from `texcount -total -brief main.tex`

Abstract - Efficient colour representation for Gaussian Splatting

This work introduces *Texture Rendering*, a new method of rendering 3D Gaussian splatting models (3DGS) by utilising the training images as textures. The technique aims to reduce the memory usage of 3DGS models, by being a more efficient representation of the scene's colour. 3D Gaussian splatting has recently emerged as a promising approach to the novel view synthesis problem, allowing faster training times and higher quality than previous state of the art NeRF techniques. The main drawback is the high memory consumption, which is a limiting factor both in training of the 3DGS models and in their deployment. The Textured Rendering technique for 3D Gaussian splatting introduced in this work completely disentangles the geometry representation from the colour representation. On my benchmark, textured rendering allows for a $\times 8$ reduction in number of Gaussian primitives in the scene, and a $\times 5$ reduction of memory usage, all without impacting the PSNR.

I perform ablations and evaluate several variants of the technique. I compare a single-sample variant, where a texture image is sampled once per pixel ray in the rendering view, and a multi-sample variant where a texture image is sampled once per intersection of Gaussian primitive and pixel ray, and find that the multi-sample version eliminates some artefacts but the single-sample variant performs better overall.

The textured rendering technique is most effective when training a 3DGS model on images with depth maps. For cases where depth information is not available, I also introduce a *Textured Training* procedure, where a 3DGS model is optimised to achieve higher quality when used in textured rendering. This is possible by ensuring that the textured rendering pipeline is fully differentiable.

All code and the dataset used can be found at
`https://github.com/Istlemin/textured-rendering-3dgs/`.

Contents

Chapter 1

Introduction

1.1 Motivation

3D reconstruction and Novel View Synthesis (NVS) are important problems in computer vision, with potential applications in VR/AR, robotics, medical imaging and more. 3D reconstruction aims to create an accurate 3-dimensional representation of an object from several 2D images, while NVS involves rendering images from novel view angles of the object. The two problems often come hand in-hand, with modern approaches to solve NVS achieving excellent results in 3D reconstruction as well.

Neural Radiance Fields (NeRFs) [1] have achieved good results for these problems by learning an implicit function representing a scene using a neural network. NeRFs have seen a large amount of research and improvement over the last years, but real-time training and rendering at a high quality has not yet been achieved.

3D Gaussian Splatting (3DGS) [2] instead approaches the problem by representing the scene as a set of coloured 3D Gaussian primitives. A differentiable rasterizer enables learning the scene by rendering and comparing against the known training views. This has achieved great results, improving state-of-the-art in quality and training cost.

A major drawback of the 3D Gaussian Splatting technique is the high memory footprint of the trained scenes. For the benchmark used in the original paper, the learnt scenes take up about 10 times larger space than the images used for training, multiple gigabytes per scene. Furthermore, to train a typical scene to high quality, tens of gigabytes of VRAM are needed. To reduce the memory footprint, there are two main opportunities for optimisation: store less data per Gaussian primitive, and use less primitives to represent the scene.

In this work, I propose a more efficient colour representation for 3D Gaussian splatting, where the images used for training can be used as textures for the model during rendering. This significantly improve both parts of the memory consumption: storing less data per

primitive and requiring fewer primitives for a given scene. With the Gaussian Primitives being colourless and only representing the shape of the scene, no colour information needs to be stored per primitive. In addition to this, a significant contributor to the number of primitives needed for a scene is high frequency colour variation. In these cases, much fewer primitives are needed to represent the lower frequency shape information, and the high frequency colour information can come from the textures instead.

One difficulty of this approach is learning a sufficiently accurate scene shape representation that avoids misalignment of textures. A proposed solution is to use depth information in addition to colour images when training the scene. For many applications, dense depth can be made available if it isn't already. For instance, in purpose build capture systems it is often possible to add a depth camera, and some modern smartphones have built in depth sensors. For cases where depth information is not easily available, I propose a texture based optimisation method, in which the 3DGS model is trained directly to give more accurate results with textured rendering.

1.2 Contributions

The main contrubtions of this work are the following:

- A novel *Textured Rendering* method for rendering 3DGS scenes with colour information taken directly from the training images.
- A method of training 3DGS with additional depth information, improving the accuracy of the textured rendering.
- An *Textured Training* technique for using textured rendering during optimisation to increase quality of the finals renders.
- Evaluation of the Textured Rendering technique, showing that it significantly reduces the memory footprint and increases the quality of renders.

1.3 Outline of the report

The structure of the report is as follows. Chapter 2 introduces some relevant background, including NeRFs and 3D Gaussian Splatting. In Chapter 3, related work regarding efficient 3DGS and texture based rendering is discussed. Chapter 4 describes the proposed textured rendering and textured training techniques in detail. In Chapter 5, the techniques are evaluated by comparing against standard 3DGS, and various ablations are performed. Finally, Chapter 6 summarises the lessons learnt and discusses limitations and future work.

Chapter 2

Background

2.1 Novel View Synthesis

Given a set of images of a scene, one might ask if it is possible to render new images of the scene from unseen view angles. This is known as the *Novel View Synthesis* (NVS) problem, a long-standing challenge in computer vision. This problem has many applications, in various creative spaces. Most significantly, it may enable users to interactively explore captured spaces, by smoothly moving around the scene. One clear example where such navigation of scenes might be of interest is in virtual and mixed reality. Other potential applications includes enhancing Google Street View coverage [4] and immersive replays in sports [5].

Approaches to the NVS problem can roughly be divided into two categories. The first, *Image Based Rendering* (IBR), consists of methods for interpolating the existing views to the unseen ones, keeping the existing views as part of the final scene representation. Interpolation can be done directly in pixel space [6] similar to video frame interpolation, using methods for finding correspondences pixels between frames e.g. optical flow. Later works instead interpreted each pixel as a 3D ray, giving rise to a 4D Light-Field, a function from ray to colour. This representation has shown success [7, 8], primarily in cases with large number of input images and small camera movements.

The second category involves predicting traditional 3D geometry and appearance representations from the observed images. This typically improves on interpolation based techniques in cases with larger camera movement and fewer images. The scene can, for instance, be reconstructed with mesh based representations [9] or voxel based volumetric representations [10, 11, 12]. With the use of a differentiable renderer, the scene representation can be learned using gradient descent. This is done by rendering the scene from an angle for which there is a known ground truth image, and computing a loss between the rendered and the ground truth images. By minimising this loss, the scene representation moves towards correctly modelling the scene. However, any discrete representations such

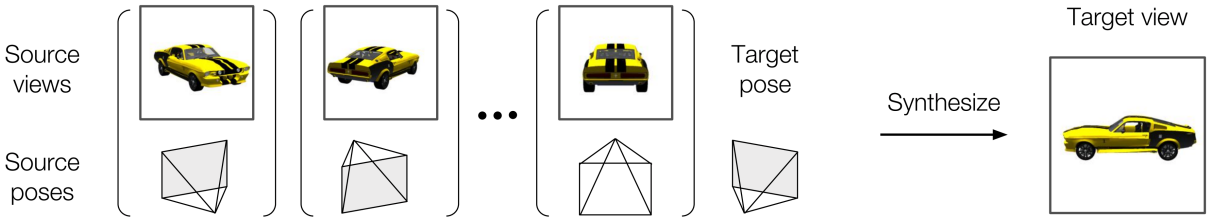

Figure 2.1: Illustration of the Multi-View Novel View Synthesis problem. A set of images from a scene together with the corresponding poses and camera parameters is given as input. The problem asks to synthesise the view of the scene from an novel target camera pose. Figure adapted from Sun et al. [3]

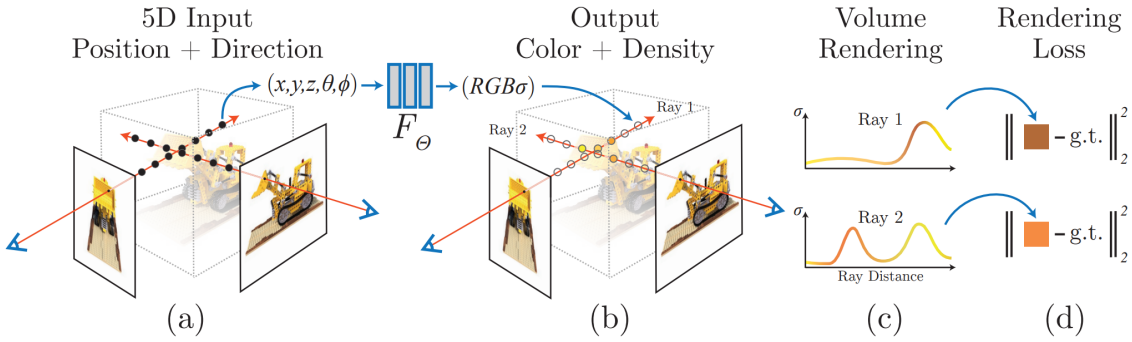

Figure 2.2: Illustration of the Neural Radiance Fields (NeRF) method for solving the Novel View Synthesis problem. Images are synthesised by sampling 5D coordinates (position and view direction) along each pixel ray (a). These are fed into an MLP, producing a colour and density at each point (b). A volume rendering technique is used to get a colour for each ray (c), which can then be compared to the ground truth colour to calculate a rendering loss (d). The scene is optimised by minimising this loss. Figure adapted from Mildenhall et al. [1]

as meshes, typically have ill-behaved gradients and are not easy to optimise. The main problem is discontinuities arising from occlusions, which motivates the usage of volumetric rendering where the entire scene including edges in inherently continuous.

2.2 NeRF

Neural Radiance Field (NeRF) [1] tackled the novel view synthesis problem by using an MLP as the scene representation. Specifically, NeRF uses a feed forward neural network to model a radiance field, a function from 3D position and view angle to density and colour. A differentiable volumetric rendering technique is then used to train the MLP to represent the desired scene. A key innovation in the approach is the usage of a Fourier basis to represent the coordinates and view angle when given them as input to the neural network. For a given point $\boldsymbol{x}$ and view angle $\boldsymbol{d}$, the colour $\boldsymbol{c}$ and density σ is calculated as

$$(\boldsymbol{c}, \sigma) = F_\Theta(\gamma(\boldsymbol{x}), \gamma(\boldsymbol{d})) \tag{2.1}$$

where F_Θ is an MLP with weights Θ and γ is a positional encoding that element wise converts the inputs to a Fourier basis:

$$\gamma(p) = (sin(2^0\pi p), cos(2^0\pi p), \ldots, sin(2^{L-1}\pi p), cos(2^{L-1}\pi p),) \tag{2.2}$$

In order to render a ray, N points are sampled along it and the colour and density of each point is calculated. The final colour of the ray is calculated using alpha-blending, with the alpha value given by the density value σ_i of the point and the length δ_i of the segment of the ray that this point represents:

$$\begin{aligned} \alpha_i &= e^{-\delta_i\sigma_i} \\ A_i &= \prod_{j=1}^{i-1}(1-\alpha_j) \\ C &= \sum_{1\leq i\leq N} c_i\alpha_i A_i \end{aligned} \tag{2.3}$$

To improve efficiency, a two-step hierarchical sampling is used. In the first step, the ray is divided evenly into segments and a single point is uniformly sampled per segment. The value A_i is calculated for each point, which represents how much the point contributes to the final colour of the ray. In the second step points are sampled from the ray by interpreting the values A_i as a piecewise constant probability density function.

This fully continuous and differentiable scene representation achieved a significant improvement compared to previous techniques. However, each scene still needed significant computational resources to train (many hours to days of GPU time). This was both due to the expensive MLP and the large number of points for each pixel it needs to be evaluated for.

The NeRF paper sparked a wave of related research which aimed to improve the computational efficiency and quality of the NeRF method. Subsequent approaches improved both the efficiency of the scene representation, some by introducing scene representations that allow fewer samples to be made per ray [13, 14] and some by speeding up the per point sampling step. Müller et al. [15] replace the positional encoding with a learnt multi-scale hash table, allowing for a much smaller MLP. Using this technique, NeRFs can be trained with minutes of GPU time instead of hours. Other follow-up work improve the quality and accuracy. Mip-NeRF [16] samples spacial Gaussian distributions from the MLP instead if points, allowing for better antialiasing. These higher quality NeRF-based methods remain computationally expensive.

2.3 3D Gaussian Splatting

My work mainly builds upon Kerbl et al.'s 3D Gaussian Splatting [2] (3DGS) technique. NeRFs showed that a suitable scene representation and rendering method is all that is needed to solve the NVS problem. 3DGS takes inspiration from this, but introduces a scene representation using coloured 3D Gaussian primitives which can be rendered with rasterisation instead of the inefficient ray marching method needed for NeRF.

Scene Representation In 3DGS, the scene is represented as the union of several coloured, anisotropic Gaussian distributions in space. Each of these 3D Gaussian distributions has an associated position $\boldsymbol{\mu}_i$, opacity o_i and a covariance matrix Σ_i. Given this, the Gaussian can be defined as

$$G_i(\boldsymbol{x}) = \exp\left(-\frac{1}{2}(x-\boldsymbol{\mu}_i)^\top \Sigma_i^{-1}(x-\boldsymbol{\mu}_i)\right) \tag{2.4}$$

The colour of the primitive is modelled to depend on view-angle, but for a given view angle the whole primitive is a single colour. The function from view angle to colour is represented using 3rd degree spherical harmonics (SH). All the per-primitive parameters – position, opacity, covariance matrix and SH coefficients – are optimised during training.

Rendering A tile-based rasteriser is used to efficiently and differentiably render the scene. For each primitive, its projection to screen space is approximated by a 2D Gaussian for rendering. Within each 16 by 16 pixel tile of the screen, all sufficiently opaque 2D Gaussians that overlap this tile are sorted by depth. Alpha blending is then applied to calculate the colour of each pixel in the tile. This follows a very similar equation to the NeRF rendering equation. For a given pixel, let $\alpha_1, ..., \alpha_N$ be the opacities of each primitive that overlaps this pixel, after accounting for covariance. The colour C of this pixel is then given by:

$$\begin{aligned} A_i &= \prod_{j=1}^{i-1}(1-\alpha_j) \\ C &= \sum_{1\leq i\leq N} c_i \alpha_i A_i \end{aligned} \tag{2.5}$$

In practice, this is computed from front to back until A_i is lower than 1%.

Optimisation With a differentiable representation and renderer, the scene can be optimized using gradient descent to match a set of training images, just as done in NeRF. Sepecifically, in each iteration a random training view is sampled. The scene is rendered from this view angle, and the resulting image is compared with the known training image. The total loss for this view is taken as the sum of an L1-loss and a SSIM loss. After each

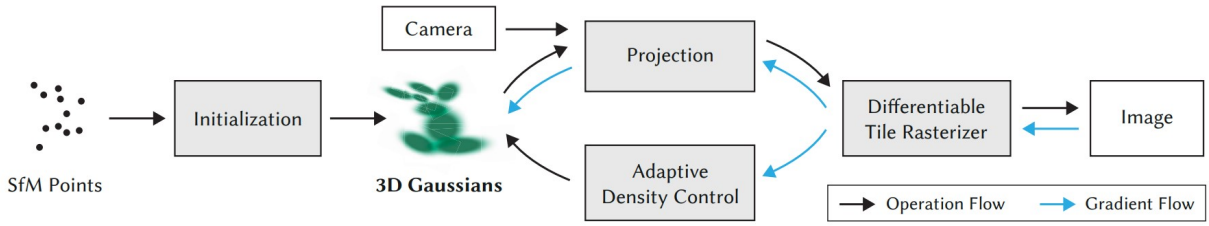

Figure 2.3: Figure adapted from Kerbl et al. [2]

image, one gradient descent step is performed on the resulting loss. The scene is typically initialised using the sparse point cloud generated by COLMAP [17, 18], which is used for camera calibration.

Adaptive Primitive Control The algorithm also involves an adaptive control of the number of Gaussian primitives. Every d ($d = 100$ by default) iterations, all primitives that are have opacity lower than a threshold ϵ_α are removed. This helps prune unnecessary primitives from the scene. On the contrary, in regions with high detail a larger number of primitives might be needed. The magnitude of the view-space positional gradient was found to be a good proxy for detecting such cases. Specifically, a primitive is duplicated if the view-space position gradient is larger than τ_{pos} (set to 0.0002 by default).

As a final step of the adaptive control, every 3000 iterations all the opacities are set to be close to 0. Gradient descent will then increase the opacity back for primitives that contribute to reconstructing the scene, while all others will be removed.

Chapter 3

Related work

3.1 Efficient 3D Gaussian Splatting

Some work has already been put into optimising the original 3DGS method by reducing the number of primitives and memory usage. The focus lies mostly on reducing the number of per-primitive parameters needed in the scene, and storing the parameters more efficiently. Papantonakis et al. [19] proposes an adaptive method for adjusting the number of SH-coefficients needed for each primitive and a codebook-based quantisation method. They also propose a pruning method that roughly halves the primitive count. Together, this achieves a $\times 27$ memory reduction. Girish et al. [20] and Liu et al. [21] uses similar techniques to achive a comparable result.

This work differs from the above mentioned by completely disentangling the colour representation from the geometry representation, instead of optimising the per-primitive colour representation. This allows the number of primitives to be further reduced, more than the previously mentioned pruning approaches.

3.2 Texture disentanglement in NeRF

The idea of separating the texture and the geometry of the scene has been previously explored in NeRFs, but with the aims of texture editing. Xiang et al. [22] models the scene using a standard NeRF network $F_\sigma(\boldsymbol{x})$ for retrieving the density at a given coordinate, but models the colour by the combination of a UV-mapping network $F_{\text{uv}}(\boldsymbol{x})$ and a texture network $F_{\text{tex}}(F_{\text{uv}}(\boldsymbol{x}), \boldsymbol{d})$. The UV-mapping network translates a 3D position into a 2D texture coordinate, and is trained together with its inverse F_{uv}^{-1} with a cyclic loss. This ensures the UV-mapping is bijective at the surface of the model. The resulting model scores slightly worse than the standard NeRF method, but enables editing of textures after training without affecting the geometry. Notably, however, the representation does not aim to improve the efficiency of the scene representation, and requires more resources

than the standard NeRF method for representing the same scene.

3.3 Depth Supervision in NeRF and 3DGS

Depth Supervision has been previously utilised in both NeRF and 3D Guassian Splatting, with the promise of allowing faster training and more accurate reconstruction in the few-image scenario. Without depth supervision, training on few (2-10) images can often lead to overfitting, where the scene looks correct from the training views but contains significant artifacts in the novel views. DS-NeRF [23] proposes a depth loss for NeRFs to deal with this problems. They optimise the probability distribution function $h(d)$, representing the probability of a photon terminating at depth d along the ray, to follow a Gaussian distribution around a ground truth depth D^* with some standard deviation σ:

$$h(d_i) = \alpha_i A_i$$
$$L_{depth} = \sum_i \log(h(d_i)) \exp\left(-\frac{(d_i - D^*)^2}{2\sigma^2}\right) \Delta_i) \tag{3.1}$$

DS-NeRF focus on using depth from the sparse COLMAP point cloud, where a standard deviation can be estimated for each point, but also support setting σ to a constant for dense depth maps with small or unknown error.

In 3DGS, depth supervision has in a similar fashion successfully been applied to allow accurate reconstruction with very few images [24]. Here, the depth for a given pixel is calculated by directly replacing colour with depth in Eq. (2.5):

$$D = \sum_{1 \leq i \leq N} d_i \alpha_i A_i \tag{3.2}$$

where d_i is the depth of primitive at position i in the blending sequence. The depth loss is then calculated as the l_1 distance from the ground truth depth:

$$L_{depth} = ||D - D^*||_1 \tag{3.3}$$

This method does work for depth supervision, but does not give an accurate depth map for models trained without depth supervision. In Section 4.1.3, I propose a modification addressing this issue. Furthermore, the focus of the depth supervision in the presented methods are different to the one of this work – they aim to reduce overfitting, while I aim to reduce the number of primitives and memory usage.

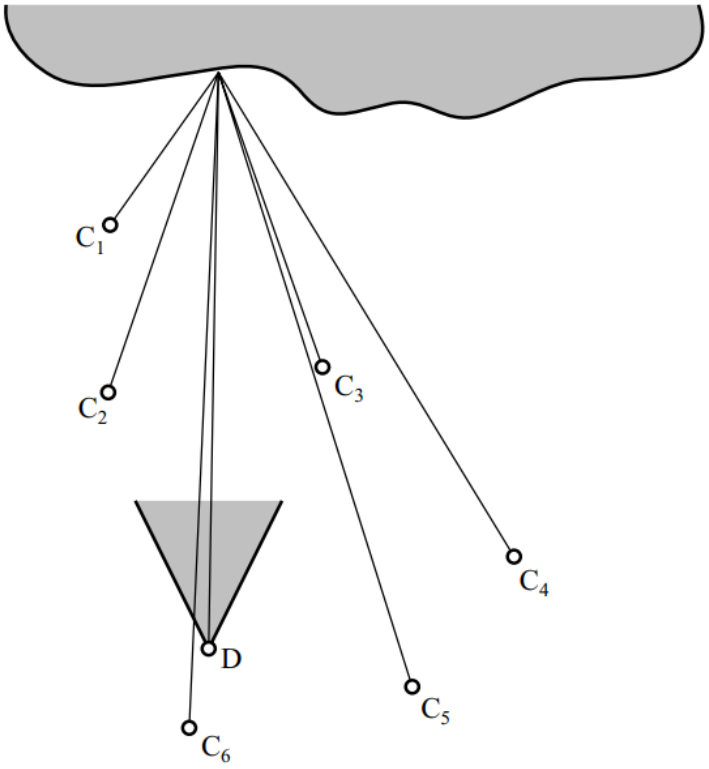

Figure 3.1: The idea behind view-dependent texture mapping (VDTM), is to render a scene from a novel view D by interpolating existing views C_i. For every camera ray from D, its intersection point with the geometry is calculated. This point is projected to all the existing views, from which colours are sampled. Figure adapted from Buehler et al [25]

3.4 Image Based Rendering

My work shares some similarities with early image based rendering works. The idea of using known views as view-dependent textures for a known geometry was proposed early in the NVS literature [26, 25]. The idea is to produce a novel view by casting pixel rays from the rendering view to find their intersection with the geometry. The intersection points are projected to the known views to extract colour, and these colours are combined using some heuristic weighting of which known views are more suitable for this pixel. This class of techniques is called *view-dependent texture mapping*, and Fig. 3.1 shows an illustration of the idea. This work takes inspiration from this approach, in the context of improving efficiency of 3D Gaussian Splatting.

Chapter 4

Design and implementation

The 3D Gaussian Splatting method has seen great success in tackling the Novel View Synthesis problem. However, the high memory needs for typical scenes is a barrier to the method's usage. A promising way of reducing the memory would be to disentangle the colour representation from the geometry representation. In the original 3DGS approach each primitive is rendered in a single colour. For many scenes, this means that many more primitives are needed to represent the high frequency colour details of the scene than are needed to represent the lower frequency geometry. In addition to this, over 80% of the data stored for each primitive is used to model the colour.

In this work, such a disentanglement of colour and geometry is achieved by not storing any colour information in the primitives, and instead sampling colours during rendering from the images used to train the scene. In effect, the training images are used as textures for the scene. This means that classical texture compression algorithms could be used at rendering time, allowing for a more efficient colour representation, while the 3D Gaussians act as a more efficient geometry representation. I call this approach *textured rendering*.

In this work, we focus two different scenarios in which textured rendering can be applied. When depth information is available, this can be used to learn a colourless 3DGS model with accurate geometry, using depth supervised training. This model can then be rendered directly with textured rendering. When depth information is not available, training a coloured 3DGS model using the standard procedure does not typically achieve accurate enough geometry for textured rendering to produce good results. Therefore I introduce *textured training*, where an existing 3DGS model is finetuned using differentiable textured rendering.

In addition to this, several variants of the approach are implemented and evaluated, including a comparison of sampling colours multiple times along a pixel ray, instead of the default of sampling a single time. Fig. 4.1 shows an overview of the single-sample textured rendering, in the case when depth information is available.

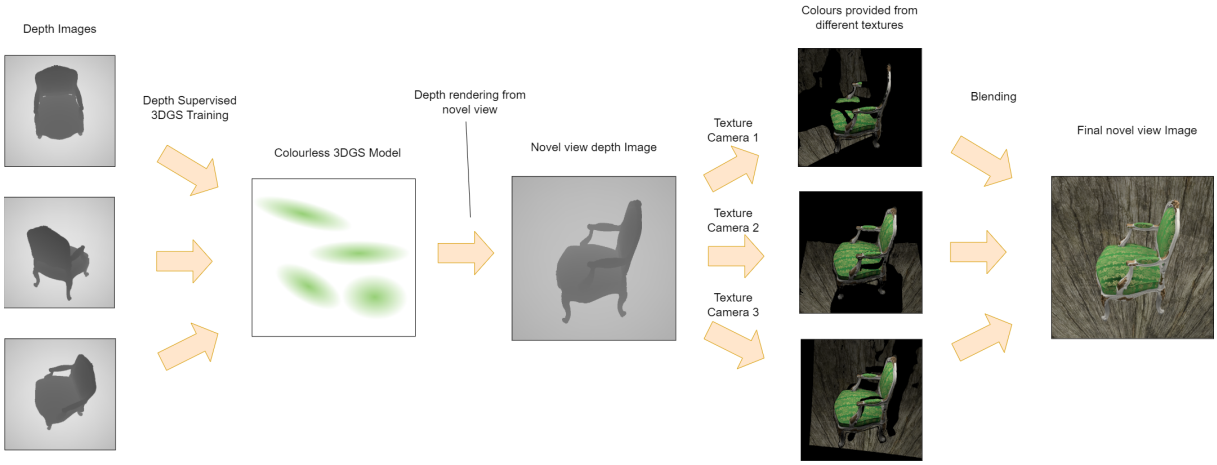

Figure 4.1: Overview of depth supervised single-sample textured rendering. Captured depth images are used to train a colourless 3DGS model, representing the geometry of the scene. To render a novel view, this model is first used to render a depth image. Given this, the 3D point on the visible surface of the scene can be calculated for each pixel in the rendered image. For each training camera, the 3D points are projected to the camera and sampled for a colour. For a given camera, some points are not visible and coloured black in the illustration above. The colours from each training camera are combined, while taking into account point visibility, to arrive at a final rendered image.

The implementation is done by extending the reference implementation [1] provided in the original 3D Gaussian Splatting paper [2]. This reference implementation is written using the PyTorch [27] framework, as well as custom CUDA kernels for the Gaussian rasteriser. Adding support for textured rendering required heavy modification of both the CUDA kernels and the PyTorch-based training code.

4.1 Textured Rendering

Two different approaches to textured rendering are implemented and compared. In the main method, which performs better in the evaluation, only a single texture sampling is done per pixel ray. I call this *Single-Sample textured rendering*, and contrast it against *multi-sample textured rendering* where a texture is sampled for each intersection of a pixel ray and Gaussian primitive. In the following, "textures", "texture views" and "texture cameras" refer to the subset of the training images and cameras used to texture the scene. Typically, all training images are used.

4.1.1 Single-Sample Textured Rendering

The proposed texture rendering pipeline starts by rendering a depth image from the novel view. If the scene is fully opaque, the geometry of the visible part of the scene is exactly determined by this depth image. In particular, for each pixel we can calculate a 3D point corresponding to the intersection of the pixel ray with the scene. Any texture view where

[1] `https://github.com/graphdeco-inria/Gaussian-splatting`

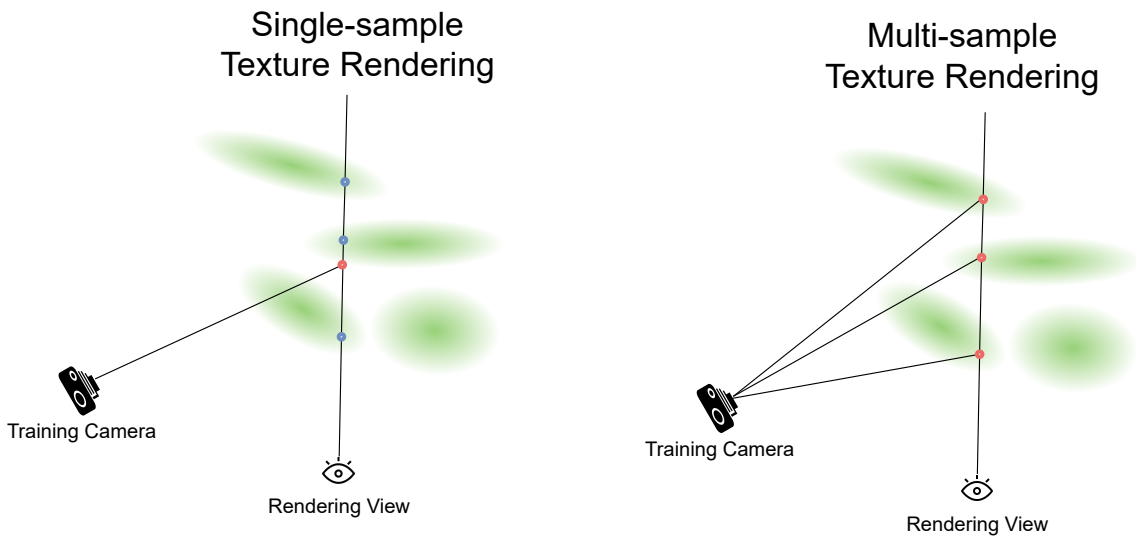

Figure 4.2: In single-sample textured rendering, a given texture is only sampled once for pixel ray at the 3D point corresponding to the surface described by the depth image. In multi-sample textured rendering, the texture is sampled once for every intersection of the pixel ray with a primitive.

some of these 3D points are visible can then provide information about their colour, and thus inform the colour of the corresponding pixels in the novel view. Therefore, for each 3D point and for each texture view, we need to determine if this point is visible from the texture view. In case a point is visible from multiple texture views, some method of deciding which view to take the colour from is needed.

Formally, let f_x, f_y, c_x, c_y be the intrinsic parameters of the view that will be rendered, and let W be the view matrix. The depth rendering procedure described in Section 4.1.3 is used to render a depth image D. For every pixel $\boldsymbol{p}$ in the rendered view, the corresponding 3D scene point $\boldsymbol{x}(\boldsymbol{p})$ in world space can then calculated as:

$$\boldsymbol{x}_{view}(\boldsymbol{p}) = D(\boldsymbol{p}) \begin{bmatrix} \frac{p-c_x}{f_x} & \frac{p-c_y}{f_y} & 1 \end{bmatrix} \tag{4.1}$$

$$\boldsymbol{x}(\boldsymbol{p}) = W^{-1}\boldsymbol{x}_{view}(\boldsymbol{p}) \tag{4.2}$$

Throughout this work, vectors such as $\mathbf{x}$ are written in bold lowercase letters. Images (such as D) and matrices (such as W) are written with uppercase letters. $D(\boldsymbol{p})$ is the value of the depth image at coordinate $\boldsymbol{p}$.

For each texture view i, an implied colour of the 3D point $\boldsymbol{x}$ can be found by projecting it to the texture and sampling the colour:

$$C_i(\boldsymbol{p}) = T_i(P_i(\boldsymbol{x}(\boldsymbol{p}))) \tag{4.3}$$

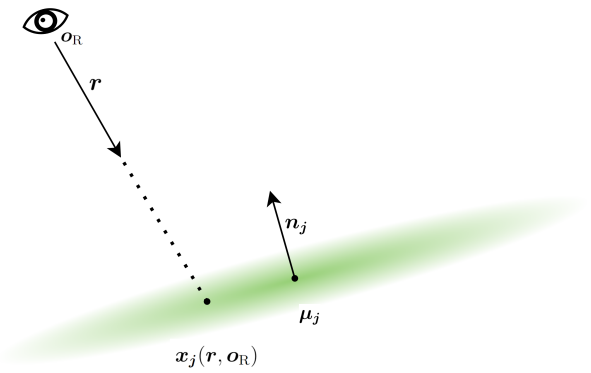

Figure 4.3: In multi-sample textured rendering, the intersection point of a pixel ray and a Gaussian primitive is found by approximating the Gaussian as a plane. The plane normal n_j is taken as the direction of lowest covariance, i.e. the shortest eigenvector of Σ_j.

where P_i is the projection function (taking a 3D point in space to a 2D image coordinate) for camera i and T_i is the texture sampling function (using bilinear interpolation). We call the ray from $P_i(\boldsymbol{x}(\boldsymbol{p}))$ to $\boldsymbol{x}(\boldsymbol{p})$ the *texture ray*.

In addition to C_i, a visibility function $V_i(\boldsymbol{p})$ is also calculated for every pixel as described in Section 4.1.4. These can then be combined into a final rendered image C_{full} as described in Section 4.1.5.

4.1.2 Multi-sample Textured Rendering

An alternative approach to textured rendering would be to sample the texture for each primitive individually, instead of applying it on the resulting surface. The colour for each pixel can then be calculated using the original Gaussian alpha-blending (2.5).

For a given pixel ray with direction $\boldsymbol{r}$ and origin at the render camera centre $\boldsymbol{o}_\mathrm{R}$, and for a primitive j, the colour is calculated as follows. First, the normal $\boldsymbol{n_j}$ of a primitive is defined as the direction of lowest variance, i.e. the eigenvector of the covariance matrix Σ_j with the lowest corresponding eigenvalue. The intersection point $\boldsymbol{x_j}(\boldsymbol{r}, \boldsymbol{o}_\mathrm{R})$ of the pixel ray and the primitive is then defined as the intersection with the plane passing through the centre $\boldsymbol{\mu_j}$ of the primitive with normal $\boldsymbol{n_j}$:

$$\boldsymbol{x_j}(\boldsymbol{r}, \boldsymbol{o}_\mathrm{R}) = \boldsymbol{o}_\mathrm{R} + \boldsymbol{r}\frac{\boldsymbol{n_j} \cdot (\boldsymbol{\mu_j} - \boldsymbol{o}_\mathrm{R})}{\boldsymbol{n_j} \cdot \boldsymbol{r}} \tag{4.4}$$

where $\cdot$ represents the dot product of two vectors.

For a texture camera i, a colour is then retrieved by projecting and sampling from the known image:

$$C_{ij}(\boldsymbol{r}, \boldsymbol{o}_\mathrm{R}) = T_i(P_i(\boldsymbol{x_j}(\boldsymbol{r}, \boldsymbol{o}_\mathrm{R}))) \tag{4.5}$$

These colours are weighted by the texture view visibility function V_i (see Section 4.1.4) before combined using Eq. (2.5):

$$V_i^{(tot)}(\boldsymbol{r}, \boldsymbol{o}_{\mathrm{R}}) = \sum_{1 \leq j \leq N} \alpha_j A_j V_i(\boldsymbol{x_j}(\boldsymbol{r}, \boldsymbol{o}_{\mathrm{R}}))$$

$$C_i(\boldsymbol{r}, \boldsymbol{o}_{\mathrm{R}}) = \frac{\sum_{1 \leq j \leq N} C_{ij}(\boldsymbol{r}, \boldsymbol{o}_{\mathrm{R}}) \alpha_j A_j V_i(\boldsymbol{x_j}(\boldsymbol{r}, \boldsymbol{o}_{\mathrm{R}}))}{V_i^{(tot)}(\boldsymbol{r}, \boldsymbol{o}_{\mathrm{R}})} \tag{4.6}$$

Where $V_i^{(tot)}(\boldsymbol{r}, \boldsymbol{o}_{\mathrm{R}})$ is the overall visibility for this pixel ray. The C_i and V_i from different texture views are combined into a final image as described in Section 4.1.5

4.1.3 Depth Rendering

For opaque scenes, the depth of a pixel ray can be said to be the distance from the camera to the first intersection point with any object. However, for a semi-transparent volumetric representation such as 3DGS, it is less clear how depth should be defined. Here, I consider depth to be the average distance to the primitives that contribute to the colour of the pixel, weighted by their contribution. This can also be viewed as the average stopping point for photon travelling along the pixel ray. Given this definition, the depth D of a pixel is calculated as

$$T_i = \prod_{j=1}^{i-1}(1 - \alpha_j)$$

$$D_{\mathrm{sum}} = \sum_{1 \leq i \leq N} d_i \alpha_i T_i$$

$$O = \sum_{1 \leq i \leq N} \alpha_i T_i$$

$$D = \frac{D_{\mathrm{sum}}}{O} \tag{4.7}$$

Where d_i is the distance from the centre of each primitive to the camera. This is different to the definition by Chung et. al. [24] use for depth regularisation, where they don't divide by the total opacity O. This causes the depth to be underestimated whenever the current scene is not fully opaque for some pixel. In section Section 5.3.4, a comparison is made between the two approaches.

The depth rendering is implemented in a differentiable manner by first employing the rasteriser to independently calculate D_{sum} and O, and then dividing the two values.

Depth Supervision As the depth rendering is differentiable, it can be used for optimisation in scenes where depth information is available. This is done by calculating a L_2

depth loss as

$$L_{depth} = ||D - D^*||_2 \tag{4.8}$$

where D^* is the ground truth depth information. In practice, ground truth depth information can be acquired using a dedicated depth camera.

4.1.4 Point Visibility

We can determine what points are visible from a given texture camera using a technique similar to *shadow mapping*. Shadow mapping is a technique for rendering shadows. In shadow mapping, a depth image of the scene is rendered from the perspective of the light source. Any point in the scene can now be projected onto the this depth map, and if it is further away from the light source than the value at the depth map, it has to be behind another object and therefore in shadow.

The same principle applies here. Before rendering a scene, we can pre-compute depth maps D_i for every texture camera i, using the technique described in Section 4.1.3. For every 3D point $\boldsymbol{x}$, projecting the point to the depth map and sampling the value gives an expected depth for a visible point from camera i:

$$D_i^{(e)}(\boldsymbol{x}) = D_i(M_i(\boldsymbol{x})) \tag{4.9}$$

Where M_i is the camera projection function for texture camera i. Comparing this to the actual depth from the point $\boldsymbol{x}$ to the camera centre $\boldsymbol{o}_i$, gives a visibility condition:

$$V_i(\boldsymbol{x}) = \begin{cases} 1 & \left| D_i^{(e)}(\boldsymbol{x}) - ||\boldsymbol{x} - \boldsymbol{o}_i||_2 \right| < \epsilon_d \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

As the scene is represented by primitives, a surface will never be fully opaque, and the exact location of the surface will slightly depend on view angle. Therefore, an inexact comparison is needed for the visibility condition. The threshold ϵ_d is a scene-dependent parameter. V_i can also be made into a continuous function:

$$V_i(\boldsymbol{x}) = e^{-\left(\frac{\boldsymbol{x}-\boldsymbol{o}_i}{\epsilon_d}\right)^2} \tag{4.11}$$

4.1.5 Blending Textures

Given the colour functions C_i and visibility functions V_i, the goal of the blending step is to produce a single colour image C_{full}, the final result of the render. This is done by computing a score $s_i(\boldsymbol{p})$ for each pair of texture camera i and pixel $\boldsymbol{p}$ in the rendered

view, and then using these scores to combine the colour information into a single image. The score aims to represent how accurate the colour provided by texture camera i is for pixel $\boldsymbol{p}$.

Scoring A score can be calculated using different heuristics. I propose two different ones here. The first one uses the *perceived texel density*, the density of texels when projected to the render view. This is proportional to the cosine similarity between the pixel ray direction and the texture ray direction, and inversely proportional to the distance to the texture camera. The score is therefore calculated as

$$s_i(\boldsymbol{p}) = \frac{||\boldsymbol{x}(\boldsymbol{p}) - \boldsymbol{o}_i||_2}{\text{CosSim}(\boldsymbol{x}(\boldsymbol{p}) - \boldsymbol{o}_\text{R}, \boldsymbol{x}(\boldsymbol{p}) - \boldsymbol{o}_i)} \tag{4.12}$$

where

$$\text{CosSim}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{||\mathbf{a}||_\mathbf{2}||\mathbf{b}||_\mathbf{2}}$$

This approximation assumes the surface at point $\boldsymbol{x}(\boldsymbol{p})$ is facing towards the render view.

An alternative heuristic is to assign the same score to every pixel for a certain texture camera, based on the distance between the texture camera and the render camera:

$$s_i(\boldsymbol{p}) = ||\mathbf{o_i} - \mathbf{o}_\text{R}||_\mathbf{2} \tag{4.13}$$

where o_i is the origin of the texture camera and o_R is the origin of the render camera. The distance heuristic achieves better scores than the texel density approach described above, and is the default in my experiments. In Section 5.3.5, the two approaches are compared.

Combining Before combining into a single image, the scores are multiplied by $V_i(\boldsymbol{p})$ to account for visibility:

$$s'_i(\boldsymbol{p}) = s_i(\boldsymbol{p})V_i(\boldsymbol{p}) \tag{4.14}$$

Given the scoring function, a final image can be constructed by taking the colour from the highest scoring camera for each pixel:

$$\begin{aligned} \text{b}(\boldsymbol{p}) &= \text{argmax}_i \left(s'_i(\boldsymbol{p})\right) \\ C_{full}(\boldsymbol{p}) &= C_{\text{b}(\boldsymbol{p})}(\boldsymbol{p}) \end{aligned} \tag{4.15}$$

Alternatively, softmax averaging can be used for a continuous approach:

$$C_{full}(\boldsymbol{p}) = \frac{\sum_i C_i(\boldsymbol{p}) \exp \frac{s'_i(\boldsymbol{p})}{\tau}}{\sum_i \exp \frac{s'_i(\boldsymbol{p})}{\tau}} \tag{4.16}$$

τ is the temperature parameter of the softmax function, and is set to $\frac{1}{4}$ for our experiments. In textured training it is necessary for the rendering function to be differentiable, and thus the softmax averaging is used.

Figure 4.4: An example case where inpainting is needed to fill in colour at the edge between the foreground (microphone) and background (brick texture). Left: The rendered depth image shows a smooth, anti-aliased edge between the foreground and background. Middle: Result from textured rendering without inpainting. The depth of the black pixels has been incorrectly estimated, and they could therefore not be provided any colour. Right: Result after nearest-colour inpainting is applied.

Invalid pixels For some pixel, there are no cameras that can provide a colour for the point, i.e. $V_i(\boldsymbol{p})$ is small for all i. This can mean one of two things. In the first case there is no view in the training set that sees the surface of the scene at this point. This can happen if the scene is non-convex, i.e. has a many holes and covered areas, and if the training set is not extensive enough. In these cases, there is no way of knowing the correct colour. In the original 3DGS method, the colour would be determined by whatever Gaussian primitives happen to overlap the unseen area, or otherwise the default background colour. As neither method proposes a solution to deals with this case, I will evaluate on scenes where this rarely comes up.

The second case where no colour may be provided, is if the depth estimate is wrong. This will lead to $\boldsymbol{x}(\boldsymbol{p})$ not lying on the surface of the model, and therefore not matching the depth images from texture cameras. In the Single-Sample Textured Rendering variant (Section 4.1.1), this commonly happens at edges of foreground objects, as the transition from foreground depth to background depth is effectively anti-aliased by the continuous Gaussian representation. The left and middle images in Fig. 4.4 show an example of this happening. The depth at the edge of the foreground object is incorrectly estimated, and no colour can be provided. To address this issue, a simple inpainting technique is used. Pixels satisfying $\sum_i V_i(\boldsymbol{p}) > 0.1$ are said to be valid, while others are invalid. For every invalid pixel, its colour is set to the closest valid pixel's colour. For Multiple-Sample Textured Rendering variant (Section 4.1.2), this is not needed as textures can be sampled twice close to edges, once for the foreground and once for the background.

4.2 Textured Training

Motivation One idea for improving textured models when depth supervision is not possible is to use textured rendering during training. That is, if the textured rendering pipeline is fully differentiable, then a loss can be computed by comparing the ground truth images to the result of textured rendering. This 3DGS model can be optimised to minimise this loss using gradient descent, similarly to how the model is originally trained. I call this *Textured Training*. Note that textured training is not stable enough to train a model from scratch, but can be used to optimise a model that has already been trained with the standard 3DGS method.

Implementation To make textured training possible, the pipeline has to be fully differentiable and gradients need to be meaningful. The depth rendering step is already differentiable, due to utilising the standard 3DGS differentiable rasterizer. The visibility masking is made differentiable by using the continuous version presented in Section 4.1.4. The operation of sampling a colour from a texture image is differentiable due to using bilinear interpolation. Finally, the blending step is differentiable if softmax averaging is used.

Note that gradients with respect to the 3DGS model can propagate to the rendered image both through the rendered depth from the rendering view, but also the rendered depth images from the texture cameras, which affect the visibility masks.

The optimisation procedure follows closely the standard 3DGS training implementation. In each step, a single training view is chosen for rendering. In textured training, the training cameras are used for both texturing and as ground truth images for optimisation. Therefore, we have to be careful to exclude the current rendering view from the available texture cameras. In order to speed up training, only the 8 closest training cameras to the rendering view are used for texturing. For simplicity, all learning rate are kept to the default values from 3DGS, but the adaptive control of primitives is completely disabled.

Multiscale Optimisation While bilinear interpolation ensures that the texture sampling operation is differentiable, it does not necessarily ensure that the gradients are well behaved. In fact, as the resulting the colour is only influenced by the four closest texels, it is easy for the optimisation to get stuck in local minima. If the texture sampling coordinates for a given point is two or more pixels away from where it should be to get the correct colour, it is not guaranteed that only a slight movement in the right direction will give a better colour than it currently has. This is a common problem in other applications where images are sampled as part of a differentiable pipeline, such as in unsupervised learning of optical flow [28, 29]. One way of addressing this problem is by simultaneously performing the optimisation at multiple scales. At a lower resolution, the distance between two points in an image will be a smaller number of pixels, and at low

enough resolutions the correct sampling location will be at most a single pixel away, and a correct gradient will thus be provided.

Inspired by this, in every step a scale s is chosen from the set $\{1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}\}$. All texture images are downsampled by a factor s before the sampling step. The image is rendered at full resolution, but after rendering also downsampled by s, before comparing against the equally downsampled ground truth training image. I also compute a combined visibility mask $V = \max sum_i V_i(\boldsymbol{p}), 1$, which is then downsampled.

By cyclicly iterating through the scales, the model both gets an opportunity to learn a high quality scene representation while getting gradient information from further away.

Loss function I propose two different loss functions to use for textured training. The first is the L1 loss between the rendered image and the ground truth image, multiplied by the combined visibility mask to account for invalid pixels ($*$ denotes elementwise multiplication).

$$L_{photo} = ||V * (C_{full} - C_{gt})||_1 \tag{4.17}$$

However, I found that optimisation is helped by explicitly encouraging fewer invalid pixels. This can be done by penalising the norm of the invalidity mask $1 - V$. However, this encourages the mask to be 1, even for points that are not visible from any texture camera. An improved loss function weighs the visibility mask by the accuracy of the colour in that place. When doing this, we can apply the loss function to the visibility mask V_i from texture camera individually, before blending.

$$L_{mask} = \sum_i \left\| (1 - V_i) * \max\left(1 - \beta ||C_i - C_{gt}||_1, 1\right) \right\|_2 \tag{4.18}$$

where $*$ indicates element wise multiplication, and β is a parameter representing the strength of the colour-dependency, by default set to 10. This loss functions ensures that pixels that are not visible from any texture camera remain invalid, as they would not be provided a accurate colour. The photometric loss and the mask loss are combined into a final loss, using the hyperparamter λ_{mask}:

$$L = (1 - \lambda_{mask}) L_{photo} + \lambda_{mask} L_{mask} \tag{4.19}$$

In my experiments, λ_{mask} is either set to 0,1 or $\frac{1}{2}$.

Chapter 5

Evaluation

The evaluation aims to determine if the proposed textured rendering technique can be used to improve the efficiency of the scene representation. Here, we consider two aspects of efficiency on a scene: number of primitives and amount of memory needed to store the scene.

Given this, the evaluation starts by presenting the main results in Section 5.2. Here, both textured rendering with depth supervision (for use when depth information is available), and textured rendering with textured training (for use when depth information is not available), are evaluated against the standard 3DGS baseline.

After this, a series of ablations are presented in Section 5.3. This includes experiments that were done in order to arrive at the best setup for the two main methods presented.

5.1 Experimental Setup

5.1.1 Datasets

As dense depth information is not available in the typical NeRF and 3DGS benchmarks [1, 16, 2], I opt to use synthetic scenes captured in Blender for evaluation. I use models from the synthetic NeRF dataset, with added backgrounds, in order to increase the detail in the scenes. For each scene, 100 camera positions are sampled from a hemisphere above the model, with all cameras facing the origin. 50 of these camera positions are taken as training set, and 50 are taken as testing set. In addition to this, I also evaluate on a more elaborate synthetic scene depicting a living room, taken from the ICL-NUIM Living Room Dataset [30]. This scene contains 100 training images and 100 test images. A sample images from these scenes can be seen in Fig. 5.1

As all scenes are synthetic, there is no sparse point cloud from COLMAP to use for initialisation, as is typically done for 3DGS. Instead, a initialisation point cloud is generated for each dataset using the depth images. All depth images are converted to point clouds

Figure 5.1: Example images from the four scenes used for evaluation. The dataset can be found from the code repository

and merged. The resulting point cloud is then downsampled to 3000 points using farthest point sampling, a downsampling technique where in every step the point furthest away from any currently selected point is selected. This ensures an even spread of points, regardless of the initial point cloud having uneven density.

5.1.2 Metrics

For evaluation, images will be rendered from the testing views and compared against the ground truth test images from the dataset. However, it can be challenging to capture the quality of a rendered image as a single number that represents the human perceived quality. Furthermore, what is considered a good render might differ by application, where e.g. colour accuracy might have high importance in some cases while correctly aligned textures are more important in others. Previous work has evaluated rendered images using three metrics: PSNR, SSIM and LPIPS. I present results using the same metrics, and briefly introduce them here.

PSNR *Peak Signal-to-Noise Ratio* (PSNR) is the simplest and most widely used of the three metrics. It is calculated as the logarithm of the ratio between the maximum pixel value and the l_2-norm of the noise. PSNR has received significant critique as a metric, and an image can be altered in a variety of ways which have little perceptual impact but greatly affects the PSNR. A small offset in colour or brightness for instance, will cause the l_2-norm to be large and thus give a low PSNR. In case of high-frequency details, a small misalignment of an image cause a low PSNR.

SSIM The *Structural Similarity Index Measure* (SSIM) aims to improve upon PSNR and be closer aligned to perceptual similarity. Instead of independently comparing pixels, it considering neighbouring pixels in a small window, and computes normalised luminance, contrast and structure values, which can be compared to the ground truth. This is less sensitive to colour perturbations, but still sensitive to misalignment.

LPIPS The *Learned Perceptual Image Patch Similarity* (LPIPS) [31] uses a convolutional neural netork to compare images. Specifically, to compare two images using LPIPS, both are passed through a VGG model [32] trained on ImageNet [32]. The internal activations in the last layers before the classifications are normalised and then compared across the two images. This gives a distance metric that aligns well with perceptual similarity.

5.2 Textured Rendering

Training of 3DGS models To compare the efficiency of the colour representation in textured rendering and normal 3D Gaussian Splatting, a number of models at different levels of detail are needed. For each of the four evaluation scenes, one model is trained with the normal 3DGS training method, and one model is trained with depth supervision, with the loss function replaced by the depth loss described in Section 4.1.3. Both are trained for 70000 iterations on the Livingroom scene, and 30000 iterations on the three simpler scenes. Optimisation parameters are kept the same as in the original 3DGS paper [2], except for the densification interval d which is increased to 1000 iterations, and the densification gradient threshold τ_{pos} which is set to 0.0003. This significantly slows down the increase of primitives, and ensures that the model has time to converge after each densification step. A checkpoint of the model is saved at regular intervals, giving a range of models at different levels of detail for every scene. Due to computational costs, densification is stopped when the number of primitives exceeds 500 000.

Rendering For each scene, each of the normal 3DGS model checkpoints is rendered with default 3DGS rendering ("3DGS"). Each of the depth-supervised checkpoints is rendered once using single-sample textured rendering ("Textured Rendering + Depth Supervision"), with visibility masking and distance scoring for blending (see Section 4.1.5).

Textured Training When depth information is not available, textured training is used to optimise the scenes before evaluation. Here, I choose a subset of the normally trained 3DGS model checkpoints, to run textured training on. For each of these, optimisation is run for 1000 iterations using L_{photo} for the Ship scene and L_{both} for all other scenes (see Section 5.3.3 for motivation). The optimised models are rendered for evaluation using the same method as for the depth supervised models.

Memory Usage In order to compute memory requirements for a model, we assume that the textured models are using all available training images for rendering, and that they are compressed using 4 bits per pixel, a standard compression ration in modern textured compression algorithms [33]. In addition to this, the textured models store 14 parameters per primitive, each a 32-bit float, while the normal 3DGS models store 59 parameters per primitive.

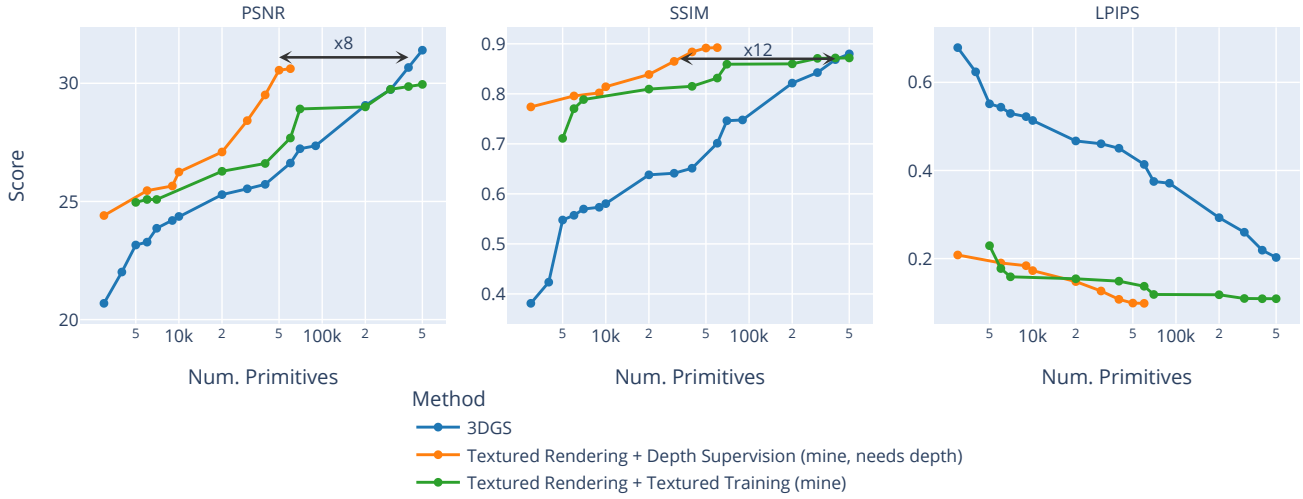

Figure 5.2: Average PSNR,SSIM and LPIPS score for different number of Gaussian primitives. For models trained using depth supervision, textured rendering shows significantly higher scores at lower number of primitives. When depth information is not available, Textured Training shows promising results as well. Note that the number of Gaussian primitives is capped at 500k for compute reasons, and that the scores for the baseline 3DGS are likely to increase with more primitives.

Table 5.1: Average score achieved with a 60,000 and a 500,000 limit respectively on the number of Gaussian primitives in the scene. At 60,000 primitives, textured rendering with depth supervision significantly outperforms the baseline.

Method	**Num. Primitives**	**PSNR** (↑)	**SSIM** (↑)	**LPIPS** (↓)	**Memory**
3DGS	60k	26.62	0.701	0.414	9.0 MB
	500k	31.40	0.880	0.203	95.0 MB
Textured Rendering					
+ Textured Training (mine)	60k	27.03	0.824	0.149	17.0 MB
	500k	29.31	0.864	0.121	31.0 MB
+ Depth Supervision (mine, needs depth)	60k	30.62	0.892	0.098	17.0 MB

5.2.1 Quantitative Results

Fig. 5.2 compares the accuracy of the original 3DGS method against textured rendering, both with textured training and with depth supervision. For each number of primitives, the presented score is the best score achieved with at most that many primitives, averaged across all scenes. We can see that textured rendering applied to the models trained with depth supervised require significantly fewer primitives than the baseline to achieve a similar or better score. An average PSNR score of 30.5 dB is achieved with only 50,000 primitives, while standard 3DGS require over 8 times as many primitives to represent the scene. A similar result is true for the SSIM score. For LPIPS, the 50k textured rendering models achieve an average score of 0.10, significantly lower than the 500k primitives baseline, which achieves an average of 0.20.

Without depth supervision, textured rendering does perform worse, but the texture training makes it outperform the baseline nevertheless. In terms of PSNR, the optimised mod-

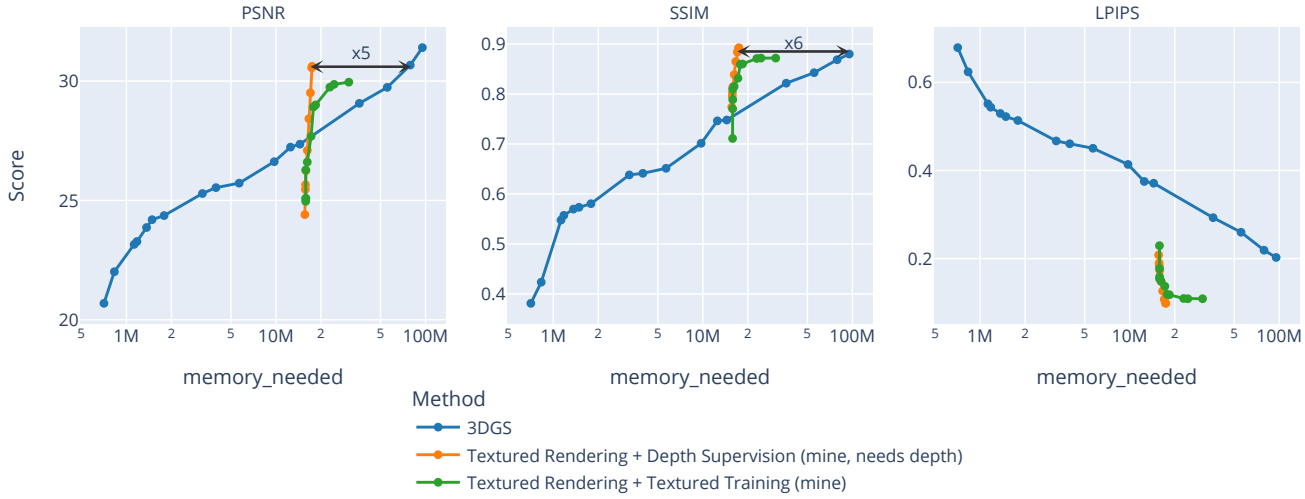

Figure 5.3: Average PSNR, SSIM and LPIPS score for different amount of model memory usage. Memory is higher when using textured rendering, as the training images used for texturing need to be stored along side the model. After accounting for this, the proposed method shows a ×5 reduction of memory usage over the baseline.

els lies 1-2 dB above the baseline for models below 200k primitives. Both in terms of SSIM and LPIPS, the optimised textured rendering models get significantly better scores than the baseline on average. To achieve a SSIM score of 0.8, my approach requires roughly ×15 fewer primitives than standard 3DGS. This indicates that in some scenarios, textured rendering can be advantageous even if depth information is not available. However, it is important to note that this advantage does depend on both the desired size of model and metric used, and a larger advantage is achieved through depth supervision.

In terms of memory costs, the textured models have a high constant cost of storing the training images, which is independent of the quality of the model. Therefore, the memory improvement is only seen for higher quality models, where the baseline 3DGS method requires a large number of primitives. Here, we can see a ×5 memory reduction for an average PSNR of 30.6 dB for depth supervised textured rendering.

Table 5.1 lists the average scores at 60,000 and 500,000 primitives.

5.2.2 Qualitative Results

Fig. 5.4 shows samples of pictures rendered with the various methods. The baseline 3DGS method with 50,000 primitives shows clear lack of high frequency detail, with the surface of the chair and the microphone looking blurry and the texture of the sofa can not be made out at all. Both variants of textured rendering perform much better, and high frequency detail is clearly visible. When depth supervision is not used, more noise is present around edges. This is due to larger inaccuracies of the depth map, and is for instance visible around the edge of the microphone and at the mast of the ship.

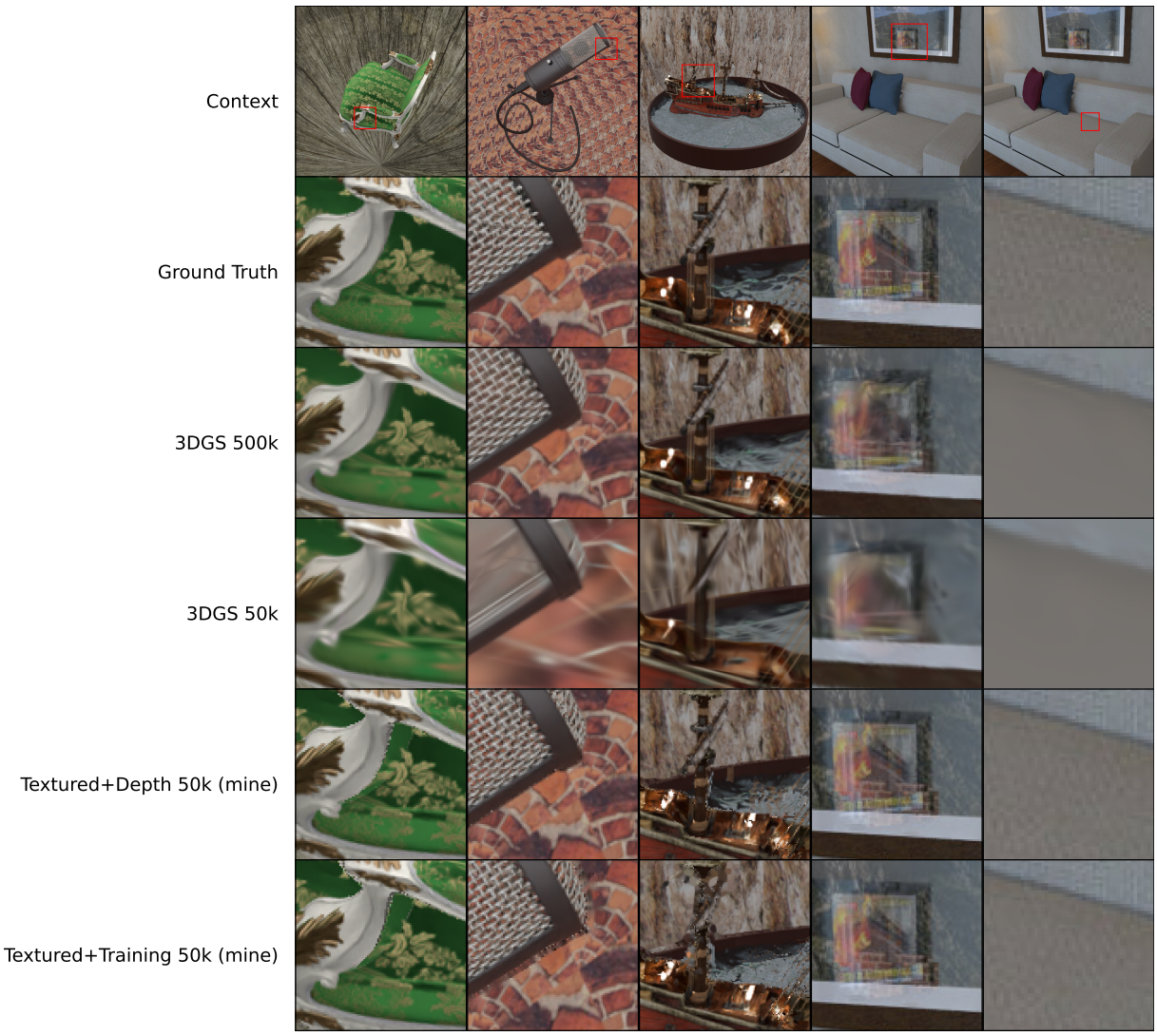

Figure 5.4: Samples of rendered images using the different methods

View-dependency Another issue with the proposed technique is strongly view-dependent features. As colour information is taken from an existing training view, there is no guarantee that the texture which the colour is sampled from will have the correct view-angle. However, due to the blending technique described in Section 4.1.5, the colour is more likely to be taken from a training view with similar angle. Therefore, when sufficiently many training views are used, view-dependent features such as reflections will be approximately correct, but sometimes slightly offset. The reflective painting in the Livingroom scene showcases this well. In the texture rendered images, the reflection of the brick wall in the painting is close to being correct but slightly too far to the right. Under the armrest of the chair in Fig. 5.4, there is a slightly brighter region, also due to view-dependency. The brighter region is not visible from the closest camera, and has to be coloured in from a slightly different angle, from which the cushion appears slightly brighter. Fig. 5.5 demonstrates this.

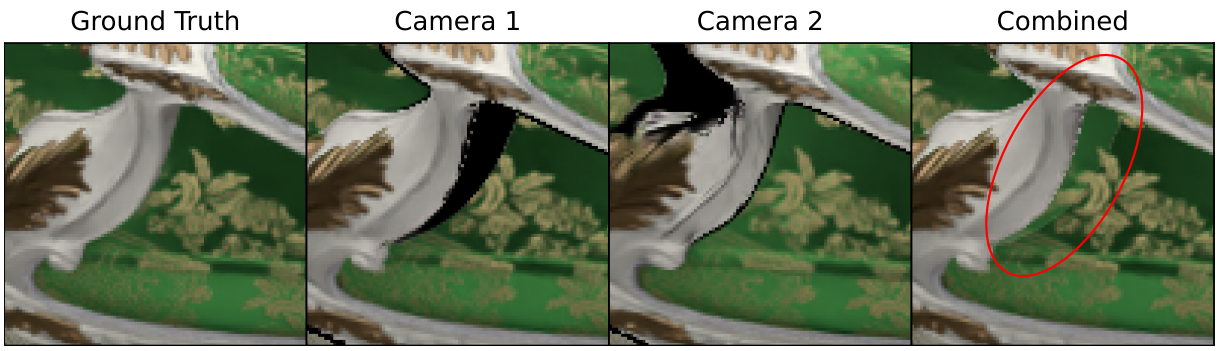

Figure 5.5: Example case where view-dependent colour causes slight visible artefact. Camera 1 and Camera 2 shows C_1V_1 and C_2V_2, the novel view coloured using each of the two closest cameras, after accounting for visibility. Camera 1 is slightly closer to the novel view, and is given priority (see Section 4.1.5) However, the area behind the armrest is not visible from Camera 1, and the colour is therefore taken from Camera 2, which happens to see region as slightly brighter due to specular reflection in the cushion.

5.3 Ablations

5.3.1 Multi-sample Textured Rendering

Table 5.2: Average scores of the final model for each scene, with multi-sample or single-sample rendering.

Method	PSNR (↑)	SSIM (↑)	LPIPS (↓)
Multi Sample Textured Rendering	28.98	0.860	0.135
Single Sample Textured Rendering	29.32	0.889	0.093

This section aims to compare the Multi-sample Texture against the Single-sample approach. To do this, the depth supervised model checkpoints described in Section 5.2 are used, once rendered with single-sample and once with the multi-sample textured rendering.

Fig. 5.6 shows the accuracy of the two approaches across various levels of model complexity (number of primitives). Table 5.2 presents the results for the final model checkpoints. We can see that multi-sample performs slightly worse than single-sample except in terms of PSNR for lower number of primitives.

The main reason for the lower scores seem to be that some surfaces appear blurrier, some examples of which can be found in Fig. 5.7. This can be explained by most surfaces not being made up by a single clear plane of Gaussian primitives, but rather a thin volume of multiple primitives that average out to make training view render to the correct depth image. As a results, when using multi-sample textured rendering a single surface might get sampled at multiple points for a single ray, and if the sampled colour for each point is slightly different, a blurred appearance is the result.

On the other side, the multi-sample method does qualitatively seem to produce less no-

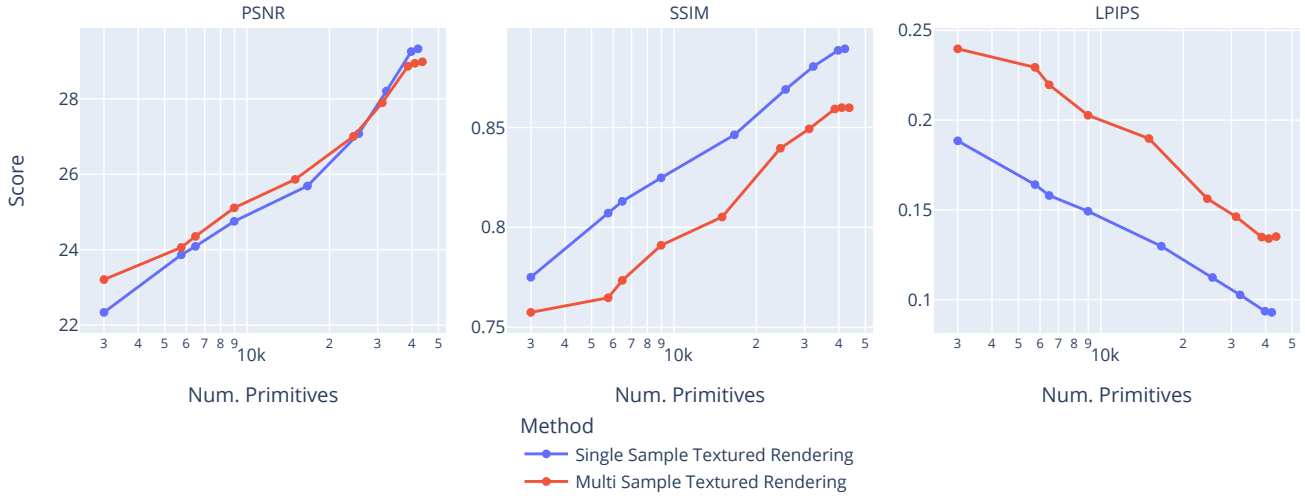

Figure 5.6: Average score of multi-sample and single-sample textured rendering, given various limits on number of primitives. Single-sample performs slightly better overall, except in terms of PSNR for low number of primitives.

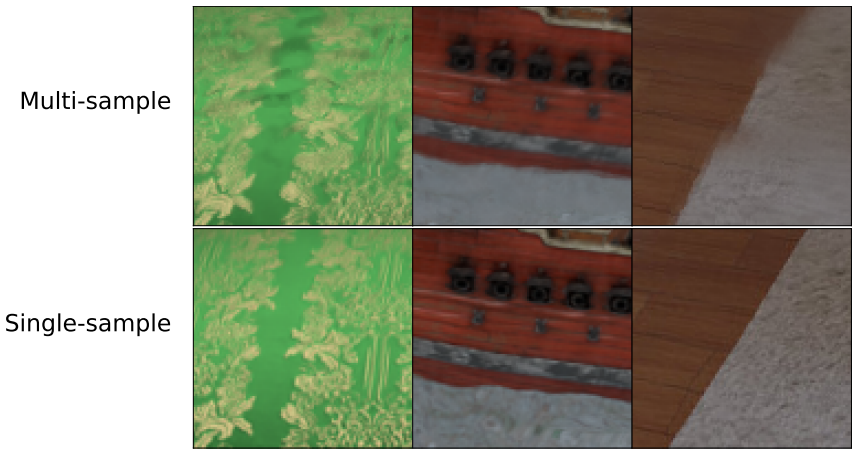

Figure 5.7: The multi-sample approach does in some cases introduce more blurring and artifacts than the single-sample approach. This can be explained by a single ray intersecting multiple Gaussian primitives for a single surface, and sampling multiple colours which are then averaged.

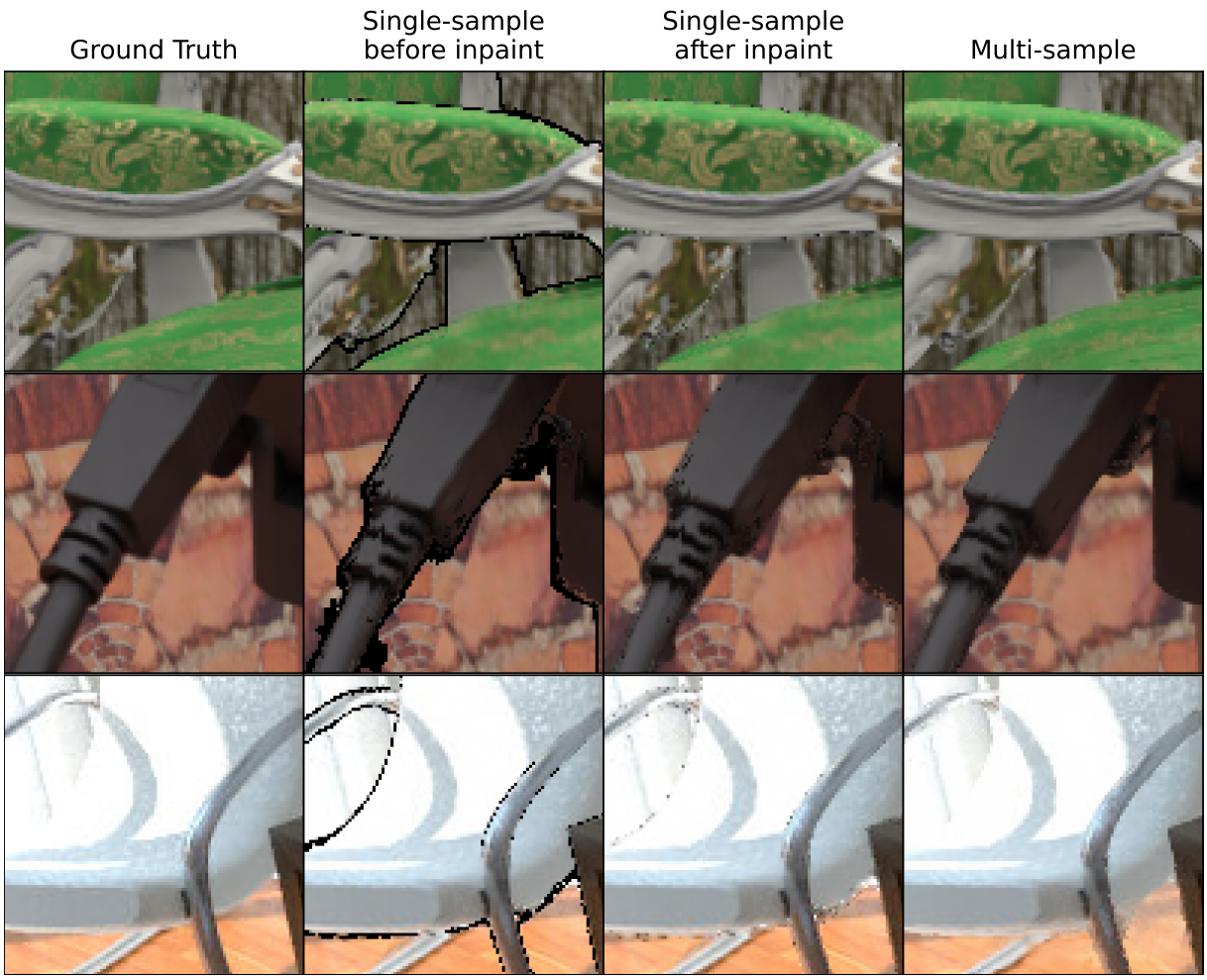

Figure 5.8: Comparison of noise at edges of objects for single-sample textured rendering and multi-sample textured rendering. In the single-sample approach, pixels at edges of objects often can not be coloured in, as seen in the second column. A simple nearest-neighbour inpainting is used to fill in the missing colours, but this leads to some visible artifacts, as seen in the 3rd column. The multi-sample approach does not require inpainting, and often leads to less noise.

table artifacts around edges and depth discontinuities. Fig. 5.8 shows examples of this. In these cases, a single depth value per pixel might not accurately describe the visible scene, as both the background and foreground can influence the colour of the pixel. For single-sample textured rendering, such pixels often cannot be given a colour from any texture, as their depth is incorrectly predicted as lying between the foreground and background. This is handled by a simple inpainting step, as described by Section 4.1.5. For multi-sample textured rendering these cases are better handled, and the inpainting step is not used. As a texture is sampled once for each primitive a ray intersects, both the colour of the foreground object and the background will be taken into account at depth discontinuities.

Figure 5.9: Average scores achievied for textured rendering on a depth supervised model, for different percentages of training images used for rendering. We see a slight decrease in scores already when using 80% of images, and with 10% the quality is significantly worse across all metrics.

5.3.2 Reducing the number of training images

Contrary to normal 3DGS, a textured rendering 3DGS model needs to store the training images alongside the Gaussian primitives. This takes up a large portion of the memory usage. However, for some scenes, not all images that were used to train the model might be needed for accurate rendering. In particular, for diffuse surfaces with low view-dependency, the surface being seen by a single training camera could be enough to provide an accurate colour. Given this, the memory usage could be reduced by only storing a subset of the training images with the model. To test this, a subset of training cameras were selected using farthest point sampling on the camera centers.

Results Fig. 5.9 shows average scores achieved when rendering with fewer images. It is clear that any decrease in number of training images used for rendering cause a small but noticeable decrease in quality. This could be due to larger portions of the scene not being visible to any training view, and thus missing colour information during textured rendering. It could be due to view-dependent artefacts being more prominent as the nearest training view is further away from the rendering view on average. To determine this, Fig. 5.10 shows the average portion of pixels that couldn't be assigned a colour, and thus requiring inpainting. This portion remains relatively low as the number of rendering images decreases. This indicates that the second option is more likely, that the decrease in quality comes from the training image used to colour a pixel is further away on average.

A consequence of these results is that the quality of textured rendering likely can be increased by including more images in training set and using these for rendering. However, this also increases memory consumption.

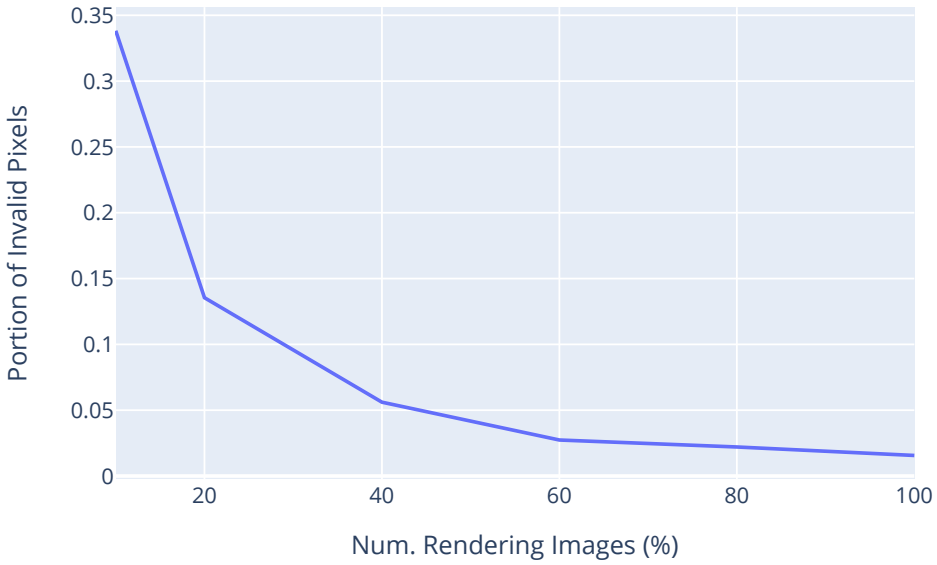

Figure 5.10: Average portion of invalid pixels, i.e. pixels that cannot be assigned a colour from any texture, for different percentages of training images used for rendering. As the percentage stays relatively low until around 20 to 40%, the degradation in quality is more likely to come from view-dependent effects.

5.3.3 Textured Training

The goal of textured training is to increase the quality of a textured model when depth information is not available. Here, I compare the result of applying textured training to directly running textured rendering on a normally trained 3DGS model. I then compare the two loss functions introduced in Section 4.2, and investigate the impact of number of training iterations.

Comparison against no textured training Using the same models as in Section 5.2, textured rendering with and without textured training are compared across various levels of detail in Fig. 5.11. The textured training significantly improves the quality across all metrics. Fig. 5.12 shows some rendered images before and after textured training. Both the textured models, with and without textured training, do a good job at capturing high frequency detail. However, without textured training, a lot of undesired noise is also present in the rendered images. This is due to slight inaccuracies in depth prediction, leading to noisy visibility maps and uneven texture mapping.

Loss functions For each scene, I use the model checkpoint after 10,000 iterations from the normal training run (see Section 5.2) as starting point. This is then optimised for 1000 iterations using textured training. In Table 5.3 I compare the results from using loss functions L_{photo},L_{mask}, and $L_{both} = \frac{L_{mask}+L_{photo}}{2}$. Surprisingly, each loss function

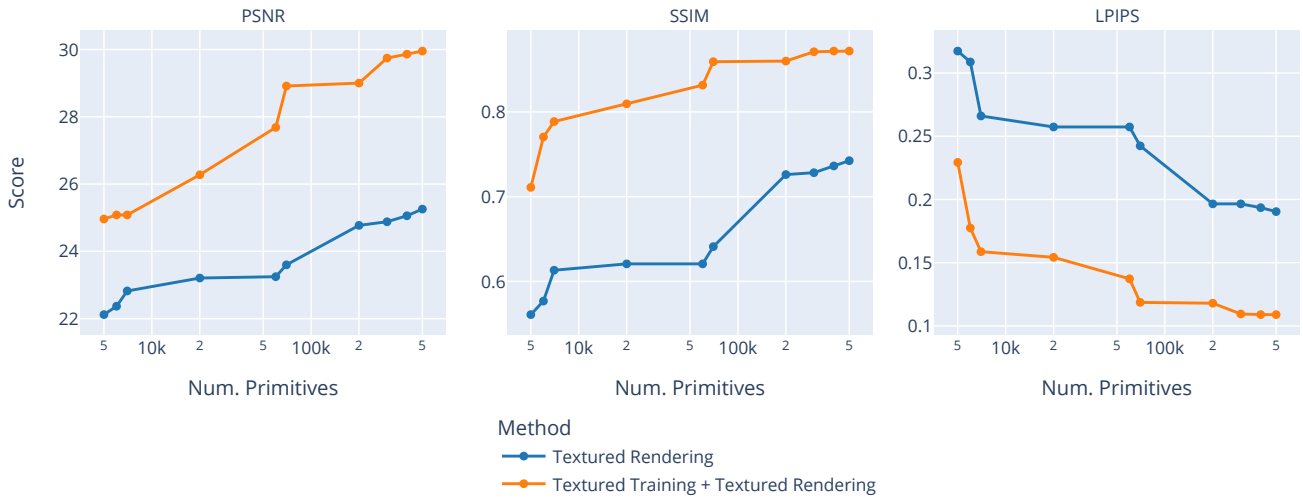

Figure 5.11: Average scores for various model sizes, comparing the standard textured rendering applied directly to a 3DGS model without optimisation, and textured rendering with optimisation

Table 5.3: PSNR (↑) of optimised models using different loss functions. Standard deviation is reported across three runs with different seeds.

	Chair	**Livingroom**	**Mic**	**Ship**
no textured training	22.22	28.01	23.06	23.40
L_{mask}	28.26±0.02	28.50±0.03	28.26±0.004	24.72±0.04
L_{photo}	28.68±0.03	28.62±0.26	28.00±0.07	**26.33±0.05**
L_{both}	**28.93±0.01**	**28.82±0.01**	**29.02±0.06**	25.30±0.00

individually does work well, and achieves significantly better results than the baseline on all scenes. Using both loss functions together achieves slightly better results than any loss function individually, except for on the Ship scene where L_{photo} performs better.

Training for more iterations Fig. 5.13 shows the average PSNR on both the test images and train images over the course of textured training, starting from the 10k checkpoint for each scene. For evaluation of PSNR, the training views are rendered in the same way as during textured training – by using the eight closest training images, excluding the itself, as textures. We see both the training PSNR and the test PSNR plateau after a couple hundred iterations, indicating that training for longer than 1000 iterations will not improve results further. Both the performance on the training set and the testing set starts to go down after roughly 1000 iterations, which is probably due to the loss functions used only being proxies of the final accuracy of the rendered images. In early experiments, optimising the L1 loss of the final, post in-painting, image was tested. This lead to unstable training, and yielded worse results than both using L_{mask} and L_{photo}. On the Microphone scene, calculating a loss using the final image gave a PSNR of 27.9 dB after 1500 iterations, while L_{both} achieves 29.0 dB.

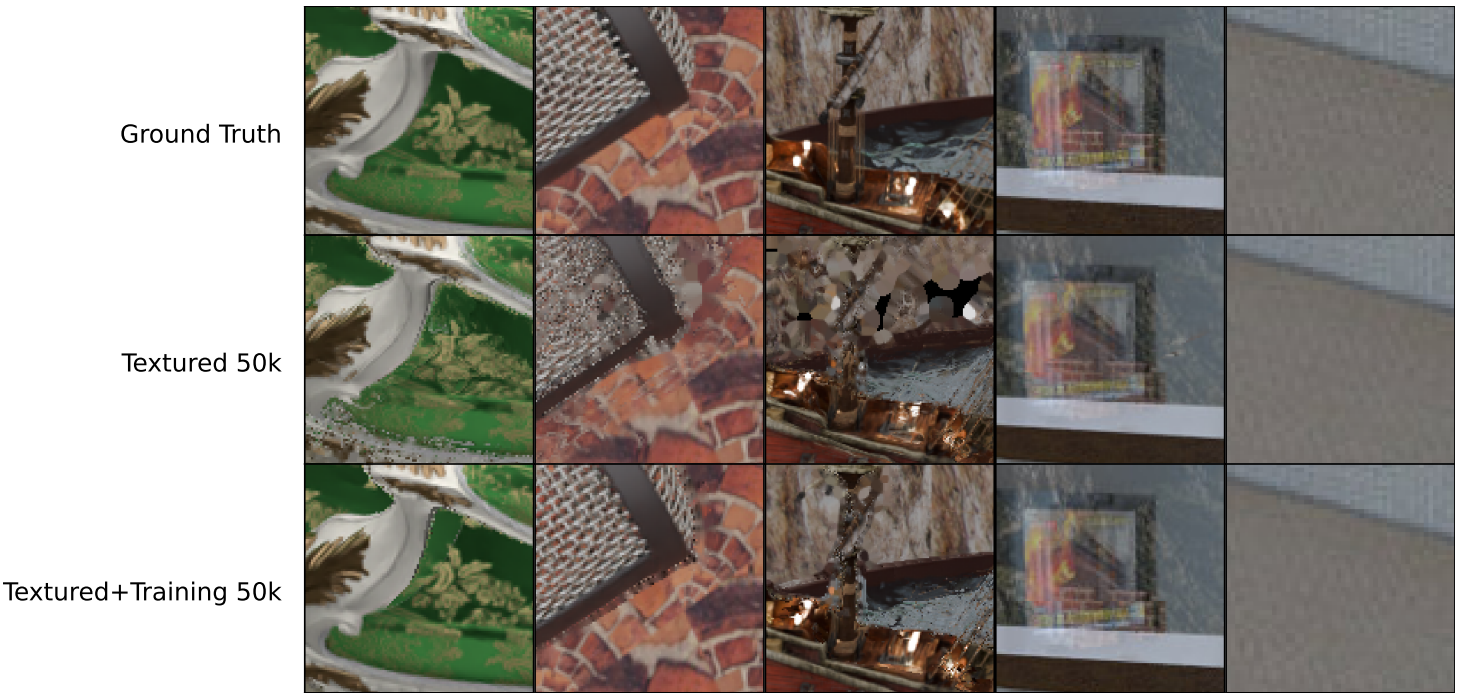

Figure 5.12: Samples of rendered images with and without textured training.

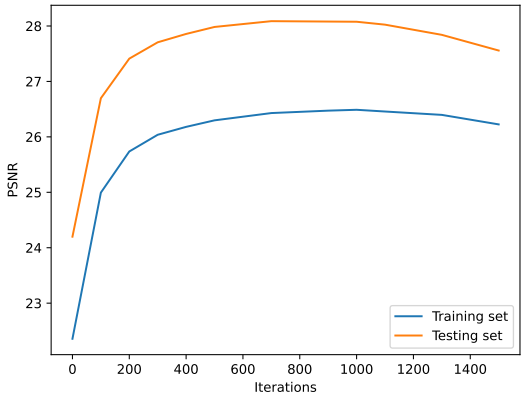

Figure 5.13: Average PSNR after different number of iterations of textured training.

5.3.4 Depth Rendering

As described in Section 4.1.3, this work uses a different definition of depth than previous work [24], as I normalise depth by the total opacity along a ray. To measure the effect of this, a comparison is made between the two techniques both in case of rendering depth images from a normally trained scene, and using depth supervision. For each scene, I consider three models: a normally trained 3DGS model, a depth supervised model trained with normalised depth, and a depth supervised model trained with non-normalised depth. Each is trained for 10,000 iterations with the parameters described in Section 5.2. The 3DGS model is rendered with both normalised and non-normalised depth, while the depth supervised models are rendered with the depth version they were trained with. For each case, the average Mean Squared Error of the resulting depth images, compared against

Table 5.4: MSE (↓) of rendered depth on training views, for different methods and scenes. Normalising the depth improves performance.

	Chair	Microphone	Ship	Livingroom
3DGS + Non-normalized Depth	6.31	5.58	1.31	0.20
3DGS + Normalized Depth	0.18	0.14	0.25	0.17
Depth Supervision + Non-normalized Depth	0.07	0.37	0.05	0.02
Depth Supervision + Normalized Depth	0.03	0.27	0.03	0.01

Table 5.5: PSNR (↑) for ablations and modifications to the point visibility and blending steps of the texture rendering pipeline

Scoring Function	Continuous Blending	Visibility Masking	Chair	Livingroom	Microphone	Ship
Distance	No	Yes	**31.89**	**32.80**	**30.24**	27.40
Distance	Yes	Yes	30.97	31.46	29.45	**27.54**
Distance	No	No	24.39	28.81	22.85	21.89
Density	No	Yes	29.86	30.49	28.19	25.63
None	No	Yes	25.99	28.62	25.95	24.03

the ground truth depth, is calculated. Results can be seen in Table 5.4. Across all scenes, and both with and without depth supervision, using the normalised depth gives a lower MSE than not normalising.

5.3.5 Blending method and visibility

In this section, I evaluate the importance of the visibility condition and the effects of the blending method. Specifically, the following parameters are compared:

- **Blending Score Function:** In Section 4.1.5, two different functions of scoring texture cameras were proposed: *Distance Scoring* and *Texel Density Scoring*. These are compared here, as well as fully ablating the score function, equivalently to setting all scores to 1 ($s_i(\boldsymbol{p}) = 1$).
- **Discrete vs Continuous Blending:** The final rendered image can either be computed by only considering the highest scoring camera for each pixel, or a continuous blending using softmax can be applied.
- **Visibility Masking:** To test the importance of the visibility masking procedure described in Section 4.1.4, the step is ablated from the pipeline. This is done by setting all pixels as visible from all texture cameras.

The default blending method uses Distance Scoring, Discrete Blending, and does accounts for visibility.

In Table 5.5, the PSNR of the best depth-supervised model for each scene is presented. We can see that Distance Scoring outperforms Density Scoring, which in turn performs much better than not scoring texture cameras at all. The continuous softmax blending also

performs worse than the discrete one, except for the Ship scene. This might be explained by Ship having the lowest PSNR and perhaps shows more artefacts, and softmax blending could slightly smooth out some of these. Lastly, we can see that the visibility masking is important for accurate rendering.

Chapter 6

Summary and conclusions

6.1 Summary and Lessons Learnt

This work aimed to decrease the memory usage and number of primitives needed for high quality 3D Gaussian splatting models. This was done by introducing a new textured rendering technique. The method disentangles the colour and geometry, and has been shown to give a more efficient colour representation. The main idea is to not store any colour information in the Gaussian primitives, but instead use the training images as textures during rendering. Two slightly different techniques were implemented. In single-sample textured rendering, a depth rendering is used to get a single 3D point for each pixel ray in the rendering view. These points can be projected to the various training images to sample a colour. In multi-sample textured rendering, one 3D point is considered for each intersection of a pixel ray and a primitive. For both version, a visibility masking is performed to take occlusions into account, before blending the colours from the various textures into a single image.

The main technique, single-sample textured rendering, achieves strong results on my benchmark. By training a 3DGS model on images with depth maps before applying textured rendering, my technique matches the result of a standard 3DGS model that has 8 times as many primitives and uses 5 times more memory. Inspection of rendered images shows that high frequency detail is more accurately rendered with textured rendering than with a standard 3DGS model of the same size. When depth information is not available, textured training can be used to improve the quality of models. This significantly improves upon just applying textured rendering to a normally trained 3DGS model, but falls short of the depth supervised approach.

Furthermore, we have shown that by altering the number of training images to use as textures, we can achieve different trade-offs between memory usage and quality of the model.

Finally, multi-sample textured rendering shows some promise by eliminating artefacts

around edges of objects, but falls short in overall metrics. This is due to slight misalignment when multiple points are sampled from what should be a single surface.

All in all, this work have demonstrated the feasibility of using textured rendering as a means to more efficient colour representation in 3D Gaussian splatting.

6.2 Limitations

The presented approach is only evaluated in rather limited scenarios. This is both in terms of only using 4 synthetic scenes, and the low maximum size of the 3DGS models, both of which are due to limited time and compute. In particular, the textured rendering models were not evaluated against the highest possible quality 3DGS models.

Furthermore, the rendering speed of textured rendering was not investigated in this work. As texture sampling is a common operation in rendering pipelines, that modern GPUs are optimised to perform, I expect that an optimised implementation would not be significantly slower than standard 3DGS rendering.

6.3 Future Work

There are several promising avenues for future work. Various techniques for memory efficient 3DGS has been proposed concurrently to this work (see Chapter 3). It could be possible to combining these with textured rendering, to determine if the techniques have complementary value.

Furthermore, the textured rendering technique has potential for improvement in various ways. Perhaps a single learnt texture together with an UV-mapping, could be used instead of deriving textures from the training cameras. This would be similar to the NeuTex [22], and would remove redundancy in overlapping textures.

Lastly, it would be interesting to see a more thorough evaluation of the technique on a varied non-synthetic dataset.

Bibliography

[1] Ben Mildenhall et al. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *CoRR* abs/2003.08934 (2020). arXiv: 2003.08934. URL: https://arxiv.org/abs/2003.08934.

[2] Bernhard Kerbl et al. "3D Gaussian Splatting for Real-Time Radiance Field Rendering". In: *ACM Transactions on Graphics* 42.4 (2023). URL: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/.

[3] Shao-Hua Sun et al. "Multi-view to Novel View: Synthesizing Novel Views with Self-Learned Confidence". In: *European Conference on Computer Vision*. 2018.

[4] Ziyang Xie et al. *S-NeRF: Neural Radiance Fields for Street Views*. 2023. arXiv: 2303.00749 [cs.CV].

[5] Sacha Lewin et al. "Dynamic NeRFs for Soccer Scenes". In: *Proceedings of the 6th International Workshop on Multimedia Content Analysis in Sports*. MM '23. ACM, Oct. 2023. DOI: 10.1145/3606038.3616158. URL: http://dx.doi.org/10.1145/3606038.3616158.

[6] Shenchang Eric Chen and Lance Williams. "View interpolation for image synthesis". In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '93. New York, NY, USA: Association for Computing Machinery, 1993, 279–288. ISBN: 0897916018. DOI: 10.1145/166117.166153. URL: https://doi.org/10.1145/166117.166153.

[7] Marc Levoy and Pat Hanrahan. "Light Field Rendering". In: *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. 1st ed. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9798400708978. URL: https://doi.org/10.1145/3596711.3596759.

[8] Abe Davis, Marc Levoy, and Fredo Durand. "Unstructured Light Fields". In: *Computer Graphics Forum* (2012). ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2012.03009.x.

[9] Tzu-Mao Li et al. "Differentiable Monte Carlo ray tracing through edge sampling". In: *ACM Trans. Graph.* 37.6 (2018). ISSN: 0730-0301. DOI: 10.1145/3272127.3275109. URL: https://doi.org/10.1145/3272127.3275109.

[10] K.N. Kutulakos and S.M. Seitz. "A theory of shape by space carving". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 1. 1999, 307–314 vol.1. DOI: 10.1109/ICCV.1999.791235.

[11] S.M. Seitz and C.R. Dyer. "Photorealistic scene reconstruction by voxel coloring". In: *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 1997, pp. 1067–1073. DOI: 10.1109/CVPR.1997.609462.

[12] R. Szeliski and P. Golland. "Stereo matching with transparency and matting". In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*. 1998, pp. 517–524. DOI: 10.1109/ICCV.1998.710766.

[13] Alex Yu et al. "Plenoxels: Radiance Fields without Neural Networks". In: *CoRR* abs/2112.05131 (2021). arXiv: 2112.05131. URL: https://arxiv.org/abs/2112.05131.

[14] Cheng Sun, Min Sun, and Hwann-Tzong Chen. *Direct Voxel Grid Optimization: Super-fast Convergence for Radiance Fields Reconstruction*. 2022. arXiv: 2111.11215 [cs.CV].

[15] Thomas Müller et al. "Instant neural graphics primitives with a multiresolution hash encoding". In: *ACM Transactions on Graphics* 41.4 (July 2022), 1–15. ISSN: 1557-7368. DOI: 10.1145/3528223.3530127. URL: http://dx.doi.org/10.1145/3528223.3530127.

[16] Jonathan T. Barron et al. *Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields*. 2021. arXiv: 2103.13415 [cs.CV].

[17] Johannes Lutz Schönberger et al. "Pixelwise View Selection for Unstructured Multi-View Stereo". In: *European Conference on Computer Vision (ECCV)*. 2016.

[18] Johannes Lutz Schönberger and Jan-Michael Frahm. "Structure-from-Motion Revisited". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[19] Panagiotis Papantonakis et al. "Reducing the Memory Footprint of 3D Gaussian Splatting". In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 7.1 (2024). URL: https://repo-sam.inria.fr/fungraph/reduced-3dgs/.

[20] Sharath Girish, Kamal Gupta, and Abhinav Shrivastava. *EAGLES: Efficient Accelerated 3D Gaussians with Lightweight EncodingS*. 2024. arXiv: 2312.04564 [cs.CV].

[21] Wenkai Liu et al. *EfficientGS: Streamlining Gaussian Splatting for Large-Scale High-Resolution Scene Representation*. 2024. arXiv: 2404.12777 [cs.CV].

[22] Fanbo Xiang et al. *NeuTex: Neural Texture Mapping for Volumetric Neural Rendering*. 2021. arXiv: 2103.00762 [cs.CV].

[23] Kangle Deng et al. "Depth-supervised NeRF: Fewer Views and Faster Training for Free". In: *CoRR* abs/2107.02791 (2021). arXiv: 2107.02791. URL: https://arxiv.org/abs/2107.02791.

[24] Jaeyoung Chung, Jeongtaek Oh, and Kyoung Mu Lee. *Depth-Regularized Optimization for 3D Gaussian Splatting in Few-Shot Images*. 2024. arXiv: 2311.13398 [cs.CV].

[25] Chris Buehler et al. "Unstructured Lumigraph Rendering". In: *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. 1st ed. New York, NY, USA: Association

for Computing Machinery, 2023. ISBN: 9798400708978. URL: https://doi.org/10.1145/3596711.3596764.

[26] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. "Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach". In: *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. 1st ed. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9798400708978. URL: https://doi.org/10.1145/3596711.3596761.

[27] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG].

[28] Rico Jonschkowski et al. "What Matters in Unsupervised Optical Flow". In: *CoRR* abs/2006.04902 (2020). arXiv: 2006.04902. URL: https://arxiv.org/abs/2006.04902.

[29] Jason J. Yu, Adam W. Harley, and Konstantinos G. Derpanis. *Back to Basics: Unsupervised Learning of Optical Flow via Brightness Constancy and Motion Smoothness*. 2016. arXiv: 1608.05842 [cs.CV].

[30] A. Handa et al. "A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM". In: *IEEE Intl. Conf. on Robotics and Automation, ICRA*. Hong Kong, China, 2014.

[31] Richard Zhang et al. *The Unreasonable Effectiveness of Deep Features as a Perceptual Metric*. 2018. arXiv: 1801.03924 [cs.CV].

[32] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[33] Jörn Nystad et al. "Adaptive scalable texture compression". In: *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*. 2012, pp. 105–114.