Single-Image 3DGS Scene Reconstruction with Geometry-Aware Priors

Machine Visual Perception Course Project Report

December 5, 2025

Information

Authors: Kacper Michalik, Radhika Iyer, Alex Loh
Group Number: 7
Supervisor: Ahmet Canberk Baykal

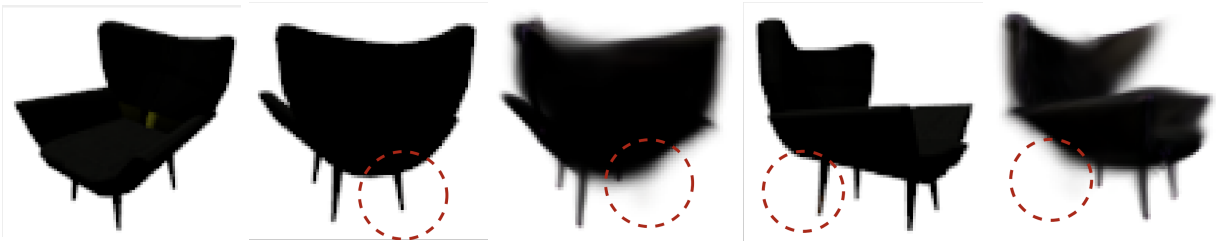
Chapter 1: Introduction and Motivation

1.1 Introduction to the problem

The advancement of 3D data acquisition, reconstruction, and rendering methods remains a fundamental and persistent open problem in computer vision. Efficient, high-quality 3D reconstruction is increasingly critical for applications ranging from augmented reality (AR/VR), autonomous devices (for example in perception or navigation in robotics or self-driving cars) to digital artistry (geometry acquisition for models in VFX, games or other digital products), driving significant research interest in this domain.
Historically 3D reconstruction has been a challenging task that requires large number of reference images and even larger amounts of compute. Advancements in computer hardware and machine-learning methods have significantly improved the efficiency and accuracy of 3D reconstruction, extending its applicability for a wider variety of tasks and hardware platforms; continuing this trend, one area of focus is made on further reducing the number of input images required whilst maintaining high quality reconstruction, thereby improving computational efficiency and reducing data acquisition hardware requirements. Namely, in 2020 Neural Radiance Fields (NeRF) [1] introduced a cutting-edge method for 3D reconstruction capable of high quality novel view synthesis. This is done by learning a continuous volumetric density and radiance function, typically using a deep learning model, which can then be queried by a ray-march for some new camera pose, enabling novel views to be synthesized. In 2023, 3D Gaussian Splatting (3DGS) [2] introduced an alternative method, offering major improvements in computational performance. Instead of an implicit function, 3DGS introduces a new explicit representation, that of a set of 3D Gaussians (called a Gaussian splat), Gaussians are volumes defined by parameters such as position, colour, opacity, rotation and scale, which can be efficiently rasterized to generate novel views. Reconstruction is typically achieved by optimizing the set of Gaussians to produce novel views with minimized loss, alternatively deep learning methods can be used to directly predict the Gaussian splat. Developments in 3D Gaussian Splatting methods have allowed for 3D scene reconstruction using few or even single RGB images. While faster than other scene reconstruction techniques and requiring only a "one-shot" pass, these approaches often suffer from challenges such as layout/scale drift, over-smooth geometry and hallucinations in occluded regions [3].

This project focuses on one recent method, Splatter Image [4], as a baseline. Splatter Image allows single or few RGB image 3DGS reconstruction. Achieved by predicting 3D Gaussians as pixels in a multichannel image; this representation reduces reconstruction to learning an image-to-image neural network, allowing the use of a 2D U-Net to form the representation. Each pixel stores the parameters for a corresponding 3D Gaussian, allowing for reconstruction in a single feed-forward pass. This overall architecture allows for a compute-efficient model. Despite its speed, Splatter Image has some issues that have been noted in related

works, particularly in reconstructing structures unseen in the input view, including for views significantly different from the source. We believe there are two reasons for this problem. The first is inherent to Splatter Image's architecture, unlike methods that utilize explicit 3D feature volumes, Splatter Image's choice of 3D reconstruction as a 2D-to-2D image translation task limits its ability to learn geometric priors, as the model lacks an internal 3D representation to reason about and resolve issues like depth ambiguities. The second is that the 3DGS prediction based on only single or few RGB image features alone does not have sufficient conditional information for Splatter Image to infer appropriate geometry information or structures, especially those that are not fully visible in the input view [5]; shown in figure 1, Splatter Image has trouble generating the occluded chair leg in 1c and 1e.



(a) Input image (b) Ground truth (c) Prediction (d) Ground truth (e) Prediction

Figure 1: Splatter Image outputs compared with ground truth taken from [5]

This project aims to address these issues, improving reconstruction quality, by first researching inferable geometry priors (such as planes, normals, visibility cues, depth, segmentation or edge maps) which can be dynamically produced for input images by existing specialized models, then proposing a lightweight augmentation for Splatter Image, allowing predicted priors to be fed alongside the RGB images, allowing them to guide and improve reconstruction, by providing necessary geometric prior information and preventing Splatter Image from having to learn these geometric features itself.

1.2 Background and related work

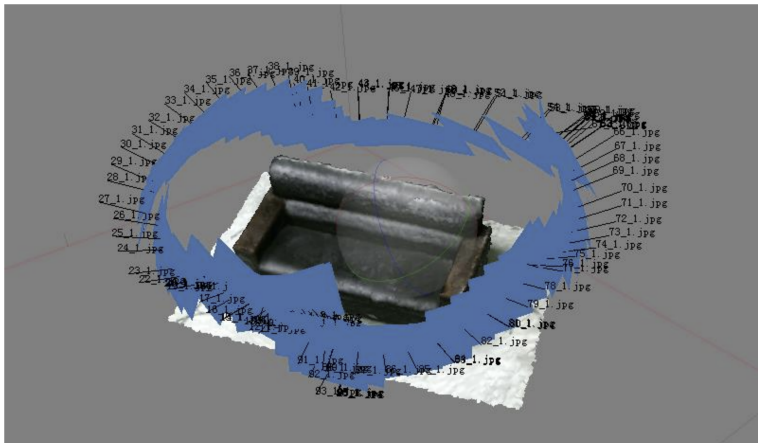


Figure 2: An example of 3D reconstruction with many overlapping images from [6]

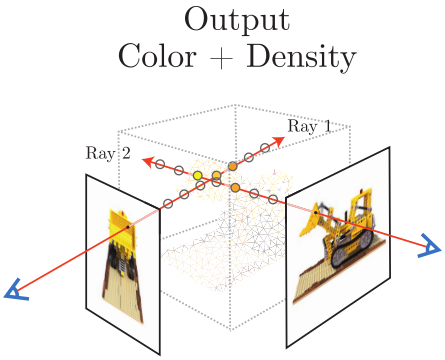


Figure 3: NeRF implicit function from [1]

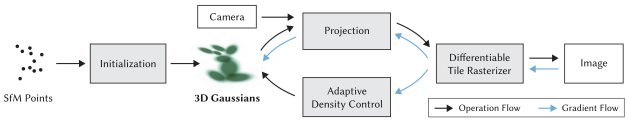


Figure 4: 3D Gaussian Splatting Process from [2]

Traditionally, 3D reconstruction has been performed by multi-stage photogrammetry pipelines, relying on explicit geometric representations such as meshes or point clouds. The industry-standard workflow begins with Structure from Motion (SfM), which matches sparse feature points across many overlapping images to estimate camera parameters and generate a sparse point cloud. This is typically followed by Multi-View Stereo (MVS) algorithms to compute dense depth maps, which are combined to generate a standard 3D mesh using techniques like Delaunay triangulation [7] or Moving Least Squares with Marching Cubes [8]. While effective for static, diffuse environments, these methods struggle significantly with surfaces such as transparent windows or reflective metals, this as they rely on photometric consistency and lack a mechanism to deal with view-dependent radiance. Additionally, these methods require large numbers of high-resolution images with substantial overlap to achieve high-quality reconstruction. This heavy data acquisition requirement means these methods create a computational bottleneck, requiring hours of processing time on high-end hardware, and are impractical in the first place without complex image acquisition setups, such as in Figure 2.

A paradigm shift occurred in 2020 with the introduction of Neural Radiance Fields [1]. NeRF moves away from explicit geometry representations to an implicit volumetric representation. In NeRF an underlying continuous volumetric scene function, represented using a deep learning model, is optimized using a set of input images with known camera poses; the input to the function/model is a single continuous 5D coordinate (spatial location and viewing direction) and the output is the volume density and view-dependent radiance at that location. Novel views are synthesized by querying 5D coordinates along camera rays for some new camera pose (ray-marching), and output colours are rendered into an image. NeRF allowed for higher quality 3D reconstruction, eventually using sparser image sets (PixelNeRF, RegNeRF), compared to traditional photogrammetry methods, achieving state-of-the-art results and becoming the gold standard for novel view synthesis [9].

In 2023 a further breakthrough in the field was achieved by the introduction of 3D Gaussian Splatting, offering a computationally high-performance alternative to NeRF. 3DGS offers a new explicit geometry representation, modelling a scene as a collection of parameterized 3D Gaussians [2]. Unlike NeRF, 3D Gaussian Splatting does not rely on a neural network to generate a scene. Instead, in the original paper, reconstruction is achieved by first initializing a set of Gaussians, either randomly or from a sparse point cloud (typically from Structure from Motion), where each Gaussian is defined by a set of parameters: position (mean), covariance (scale and rotation), opacity, and colour (represented using Spherical Harmonics to allow for view-dependent radiance). Then an optimization process is run that first adjusts the Gaussians' parameters to minimize the error between rendered images and the ground truth; and secondly performs dynamic management of the density of Gaussians within the scene, by splitting, cloning or pruning Gaussians in an interleaved manner; namely Gaussians in under-reconstructed areas are cloned, Gaussians with high variance are split, and Gaussians of low opacity and of excessive size are pruned. Finally, for rendering, the Gaussians are rasterized using a custom tile-based rasterizer to produce resulting views, allowing millions of Gaussians rendered in real-time, allowing for real-time novel view synthesis. While the original method relies on per-scene optimization to generate 3D Gaussian Splats, recent works have begun using deep learning methods to predict sets of Gaussians directly, with most recent developments allowing for 3D scene reconstruction using few or even single RGB images.

Since the introduction of 3DGS, a number of deep learning architectures and processing pipelines based on

the method have been developed to find the most accurate and efficient implementation capable of producing high quality 3D Gaussian Splats. Recent examples include ExScene [10], Wonderland [11], F3D-Gaus [12], Gauss VideoDreamer [13], TGS [14] and Splatter Image [4].

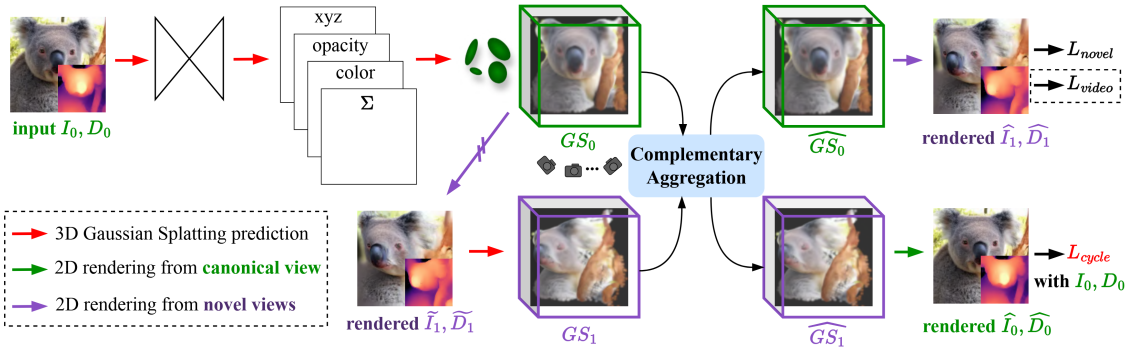


Figure 5: F3D-Gaus Framework from [12]

F3D-Gaus was proposed to help pixel-aligned Gaussian Splatting generate plausible novel viewpoints by introducing a cycle-aggregative strategy. As shown in figure 5, given a single RGB image I_0 and depth map D_0, the model directly feeds them forward to output the pixel-aligned Gaussian Splatting representation GS_0. The model then renders the image $\tilde{I}_1$ and depth map $\tilde{D}_1$ for the novel view, and then outputs its corresponding Gaussian Splatting representation GS_1. GS_0 and GS_1 are then aggregated to produce images for supervision. By enforcing this cycle consistency through aggregation, the model naturally learns through this self-supervised approach to extrapolate across views by fusing multiple representations, enhancing 3D coherence during inference. This helps to ensure that 3D representations from novel viewpoints are both aligned with and complementary to those from the original viewpoint. After complementary aggregation, F3D-Gaus then applies a video in-painting model [15] to the rendered images and depth maps to correct inconsistencies in geometry and texture caused by large viewpoint shifts, resulting in more reliable and visually consistent outputs. While the source code and a pretrained model is available, the model would require 13 days of an A100 GPU to train if any updates were made, so we deemed this unsuitable to work on.

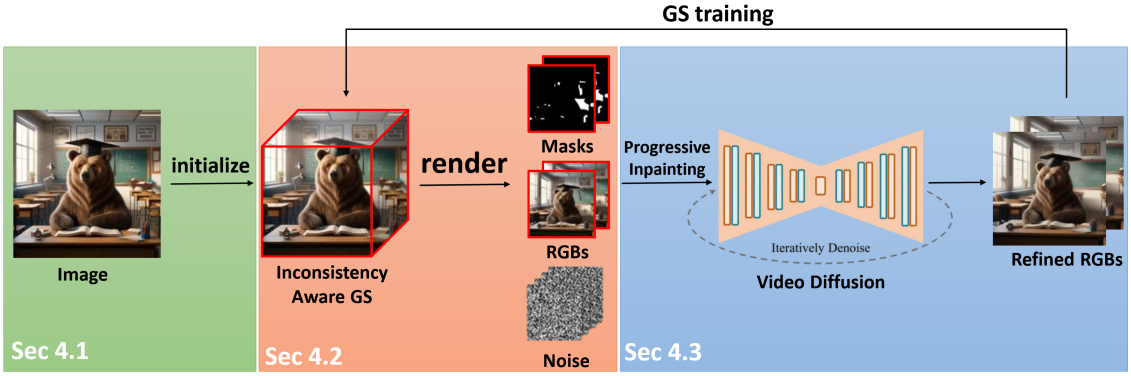


Figure 6: GaussVideoDreamer Framework from [13]

For an input image, GaussVideoDreamer first initializes a coarse video sequence depicting occluded regions of the input image under smooth novel view trajectories and inconsistency-aware Gaussian Splats. After that, at periodic optimization intervals, the model renders all viewpoint images and their corresponding inconsistency prediction masks from the Gaussian Splats. The masks and rendered images then guide a video diffusion model to perform progressive inpainting to refine the video sequence, which then in turn optimizes the Gaussian splatting representation and gradually generates better novel view images. However, the generation quality of GaussVideoDreamer is mainly limited by the video diffusion prior (AnimateDiff

[16]), which was adapted from an image diffusion model lacking dedicated video inpainting training. As such, when generating videos it may result in frame inconsistencies in longer sequences, progressive colour shifts and high-frequency detail degradation, which reduces novel view reconstruction quality. There was also no publicly available implementation of this method. In addition, since a proprietary evaluation dataset of 20 generated single-view images with corresponding text prompts was used in the paper, it would be difficult to compare it with other implementations. Therefore, we also decided not to explore this method.

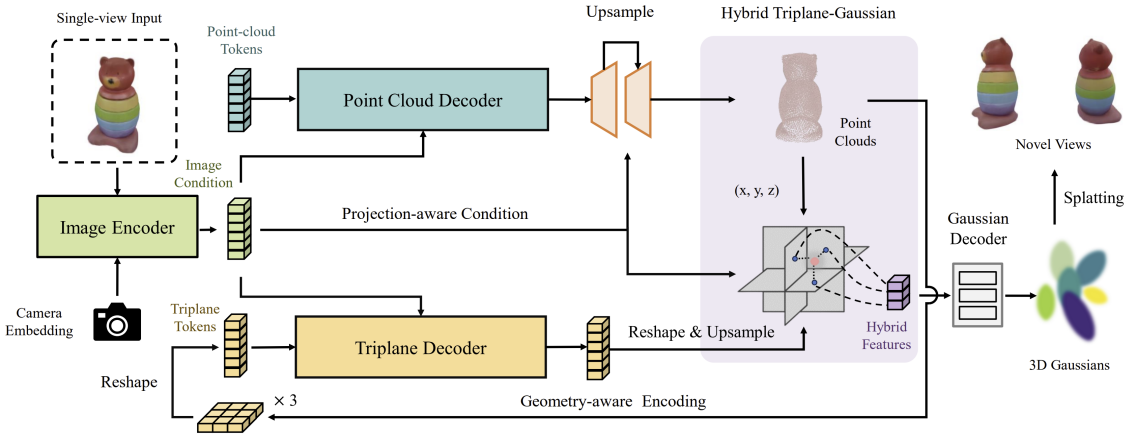


Figure 7: TGS Framework from [14]

In a triplane representation [17], explicit features of the image are aligned along three axis-aligned orthogonal feature planes. We can then query any 3D position by projecting it onto the three feature planes, retrieving the corresponding feature vector via bilinear interpolation and aggregating the three feature vectors via summation. An additional lightweight decoder network interprets the aggregated 3D features as colour and density. This representation is fast and scales efficiently with resolution by keeping the decoder small and shifting the bulk of the expressive power into the explicit features. TGS [14] uses this representation in Triplane-Gaussian, a new hybrid representation. As shown in figure 7, TGS first encodes the input image into a set of latent feature tokens, which are then passed into a point cloud decoder and a triplane decoder. A point cloud and a triplane can be de-tokenized from the output of these decoders. After the point cloud decoder, TGS adapts a point upsampling module to densify the point cloud and utilizes a geometry-aware encoding to project point cloud features into the initial positional embedding of triplane latent. Lastly, 3D Gaussians are decoded by the point cloud, the triplane features and the image features for novel view rendering through Gaussian splatting. While an official implementation of this is available at https://github.com/VAST-AI-Research/TriplaneGaussian and a pretrained model is also available, the model is only trained on the Objaverse-LVIS dataset and the training time and compute needed to adapt this implementation also made it unsuitable for our project.

1.3 Overview of the idea

Splatter Image is a model introduced in late 2023, whose main highlights are its computational efficiency, and ability to perform one-shot 3D scene reconstruction, whilst maintaining high quality reconstruction capabilities. This combination make it a highly desirable model. Nonetheless, the model's occasional problems such as layout/scale drift, over-smooth geometry and poor quality hallucinations in occluded regions, pose a serious problem in its usage and wider adoption. As stated in the introduction, we believe the cause of these reconstruction failures to be twofold; first is inherent to Splatter Image's architecture, the models choice of 3D reconstruction as a 2D-to-2D image translation task limits its ability to learn geometric priors, as the model lacks an internal 3D representation to reason about said priors. Second is our belief that single or few RGB images simply does not have sufficient conditional information for Splatter Image to infer adequate geometric information about 3D scenes such as object structures, especially those that are not fully visible in the input view. Modern machine learning now offers highly specialized models capable of performing

high-quality monocular prediction of geometric features from RGB images. These models such as MiDaS and Depth Anything, are trained on vast datasets and have tailored architectures to reason about relevant 3D information; as such they store much valuable latent geometric knowledge, which is valuable in situations exactly like those where Splatter Image struggles, namely when trying to accurately hallucinate occluded regions or trying to understand complex 3D surfaces and structures. The ready availability of these models allows us to exploit them to gain additional sources of high quality geometric information that can be fed into Splatter Image, solving both of our supposed problems simultaneously: preventing the Splatter Image model from needing to implicitly learn geometric features, whilst also providing rich 3D information required for high quality reconstruction.

As such we believe dynamically producing and feeding these priors will effectively help guide Splatter Image in 3D reconstruction, resulting in superior reconstruction quality and strongly reducing the frequency of reconstruction failures. All achieved whilst preserving Splatter Image's key features such as its computational efficiency, as it requires minimal architectural changes to accept these new priors in its input; altogether making Splatter Image a more competitive one-shot 3D reconstruction model. Furthermore, this project gives us an opportunity to perform an ablation study to see which priors are most significant in resulting 3D reconstruction quality, this study may provide useful guidance for other projects; namely helping researchers/developers decide on model architectures that work with optimal sets of geometric priors; or helping those that may be interested in creating similar adaptations/pipelines as us.

Chapter 2: Method

2.1 Baseline algorithm

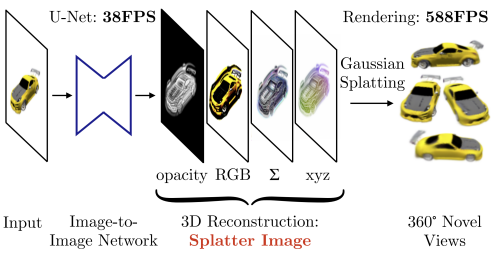


Figure 8: Overview of SplatterImage[4]

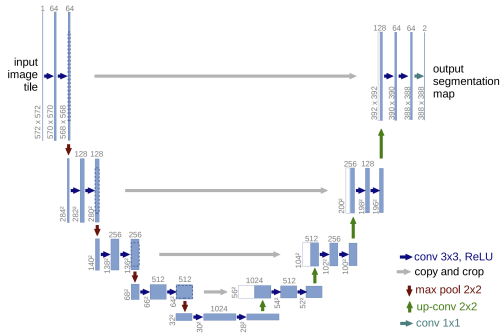


Figure 9: U-net architecture[18] that Song U-Net[19] is based on

Splatter Image uses a standard image-to-image neural network architecture to predict a Gaussian for each pixel of the input image, generating the resulting Splatter Image as output. For this it uses a U-Net, as these have demonstrated excellent performance in image generation [20], and their ability to capture small image details helps to obtain higher-quality novel view reconstructions.
As shown in figure 9, a U-Net model architecture consists of a contracting path (left side) and an expansive path (right side). The contracting path aims to capture context and follows the typical architecture of a convolutional network, which consists of the repeated application of convolutions, each followed by an activation function and a max pooling operation for downsampling. At each downsampling step we increase the number of feature channels. Afterwards, each step in the expansive path consists of an upsampling of the feature map followed by a convolution that decreases the number of feature channels, enabling precise localization.
For the U-Net used in Splatter Image, learning to predict the Splatter Image can be done on a single GPU using at most 20GB of memory at training time for most single-view reconstruction experiments (except for Objaverse, where 2 GPUs were used and 26GB of memory was used on each). Most of this neural network

architecture is identical to the SongUNet used in [19], with the exception that the last layer is replaced with a 1×1 convolutional layer with $12 + k_c$ output channels, where $k_c \in \{3, 12\}$ depending on the model configuration. The output tensor codes for parameters that are then transformed by non-linear functions to obtain the final Gaussian splat parameters, such as opacity, offset, depth, scale, rotation and colour respectively. The original Gaussian Splatting fast tile-based rasterizer implementation from [2] is used to perform rasterization to generate 360° novel views of the 3D reconstruction defined by the predicted Gaussian splat.

2.2 Algorithm improvements

2.2.1 Model Improvements

The first modification to Splatter Image's architecture is to allow the model to be initialised with a dynamic number of input channels, as opposed to the standard 3 for RGB; the required number of channels is calculated by the `GaussianSplatPredictor` module at runtime, based on the supplied training configuration. This allows models using the desired combination of additional priors to be instantiated.

The second modification is support for multimodal priors. While priors like depth or normal maps, which are images themselves, can be appended as additional channels to RGB images to generate the final model input, structurally different priors cannot simply be appended in the same way. For example, modern segmentation models, like some of those we researched, are capable of producing classifications (and instance IDs in the case of instance or panoptic segmentation) for identified segments. These classifications are typically in the form of strings or vector embeddings (which can be produced from strings regardless). These vector embeddings do not have matching dimensions to the RGB images and thus cannot be added as an extra channel in the model input. There are a variety of ways these multimodal priors can be provided to the model, one method is broadcasting; in broadcasting multimodal inputs are appended to an image by replicating them across every pixel. Since the multimodal input is typically a 1D vector (say a scalar or vector embedding) and the image is a 3D tensor (Height $\times$ Width $\times$ Channels), the vector is replicated at every single pixel location by being concatenated along the channel dimension. This method is poor due to its computational cost and massive data redundancy: the network is forced to process and store the exact same values many times, which wastes GPU memory and compute. Alternatively, Feature-wise Linear Modulation (FiLM) offers a significantly more efficient method for multimodal data input for U-Nets and Convolutional Layers. Instead of multimodal data being inserted into the input image, FiLM injects this data by modulating the intermediate feature maps of the network. FiLM layers are inserted into the model at specific points, for example between a convolution and a ReLU activation or at the end of a UNet block, these FiLM layers contain a generator, which is a separate, small neural network (like an MLP or RNN) which takes only the multimodal data (such as the segmentation embedding, which we will call $\mathbf{z}$) as input, and generates two output terms, γ (scale) and β (shift). The FiLM layer then takes the intercepted feature map and applies the FiLM equation to it:

$$\hat{F} = \gamma(\mathbf{z}) \cdot F + \beta(\mathbf{z}) \tag{1}$$

This modulated feature map is finally passed on within the network. The method is highly efficient, only requiring a multiplication and addition be performed to terms in the feature map, and allows the models image input to remain unchanged. As such this strategy of inserting FiLM layers into Splatter Image is how we achieve multimodal data support.

2.2.2 Low-Rank Adaptation (LoRA)

To address the computational constraints of training the full U-Net architecture, we integrated Low-Rank Adaptation (LoRA) [21] directly into our GaussianPredictor, to instead allow for fine-tuning.
Adapters are small modules placed after the frozen modules we wish to adapt, such as linear and convolutional layers. The adapter accepts the same input dimension as the original layer and produces the same output dimension. This allows the output from the adapter to be summed element-wise with the frozen layer's output.

Instead of learning a new large weight matrix for these adapters, the weight update is decomposed into two smaller low-rank matrices. For a weight matrix $W \in \mathbb{R}^{d_{out} \times d_{in}}$, the update is defined as $W + \Delta W$, where ΔW is factored into:

$$A \in \mathbb{R}^{r \times d_{in}} \quad \text{and} \quad B \in \mathbb{R}^{d_{out} \times r}$$

Here, we see that $r \ll \min(d_{in}, d_{out})$, where r is the rank hyperparameter (detailed below). During training, only the parameters of A and B are updated, while the original weights W remain frozen.
For `Conv2d` layers, we treat the kernel $W \in \mathbb{R}^{C_{out} \times C_{in} \times k \times k}$ as a flattened matrix of shape $C_{out} \times (C_{in} \cdot k \cdot k)$. Flattening allows the low rank adapter to process spatial neighbourhoods, and the flattened representation is decomposed into B and A, allowing us to apply LoRA to the full spatial kernel. By including the spatial dimensions ($k \times k$) in the decomposition, the adapter can learn spatial feature refinements, rather than being limited to channel-wise linear projections. This approach follows the implementation found in `loralib` [22].
During the forward pass, the input x is processed by both the frozen weights W and the LoRA branch:

$$h = Wx + \frac{\alpha}{r} BAx \tag{2}$$

where $\frac{\alpha}{r}$ is a scaling factor [22]. This scaling normalises the updates across different rank choices, reducing the need to re-tune the learning rate when r changes. We utilise three hyperparameters to control this adaptation:

- Rank (r): The rank of the low-rank matrices A and B. Higher ranks increase the number of parameters and the capacity of the adaptation, but also increase computational cost
- Alpha (α): A scaling factor applied to the LoRA update during the forward pass. The update is scaled relative to the weights that have been frozen from the pretrained model.
- Dropout (p): The dropout probability applied to the LoRA layers during training. This randomly disables activations, which aims to prevent overfitting.

Following [22], A uses Kaiming uniform intialisation, and B is initialised to zero. This ensures that $\Delta W = 0$ at the start of training, which preserves the behavior of the pre-trained model initially.
During inference, the LoRA weights can be optionally merged into the base weights by computing $W_{merged} = W + \frac{\alpha}{r} BA$, eliminating the separate branch computation and reducing overhead for efficiency.

2.3 Implementation details

All code associated with the project can be found in the following repositories:

3DGS-priors (Top level repository for the project): `https://github.com/Kacper-M-Michalik/3dgs-priors`
Splatter Image Fork: `https://github.com/Kacper-M-Michalik/splatter-image`

Generated datasets and model weights can be found in the following repositories:

Datasets with Predicted Priors: `https://huggingface.co/datasets/MVP-Group-Project/srn_cars_priors`
Pretrained Models: `https://huggingface.co/MVP-Group-Project/splatter-image-priors`

2.3.1 Planes and Normal Maps Exploration

We considered providing the model with structural information to be one of the most likely avenues of improvement. These structural priors were considered in two flavours, in the form of predicted scene planes and scene surface normal maps.

When researching plane prediction, we reached the conclusion that this flavour would in fact be unlikely to help guide reconstruction. For example teddy bears (as seen on the CO3D[23] Teddybears dataset) have complex, convex shapes as shown in 10, lacking dominant planes on their surface, using a planar prior might confuse the network, causing it to flatten the bear's features or causing poor quality hallucination; as such we decided against using planes as a prior.

Figure 10: Example of CO3D Teddybears dataset from [23]

A much more favourable option were normal maps. Normal maps store surface normal data as RGB colour information, showing the orientation of a surface on a per-pixel level, we considered this to be an excellent prior as it supports both complex shapes such as teddy bears, but can equally well describe a planar surface. Hence, we selected this prior as a prime candidate that could improve the models' 3D surface reconstruction and help guide accurate hallucination in occluded regions. We take our ground truth images from a dataset (`https://github.com/Xharlie/ShapenetRender_more_variation`) provided by [24] which contain higher resolution images of ShapeNet[25] models. Each RGB image of a ShapeNet model is paired with its corresponding depth map, normal map and albedo map as shown in figure 11. We feed these images into the normal map generation models and compare against the ground truth normal maps, using Pixel Based Visual Information Fidelity as a metric to evaluate their performance.

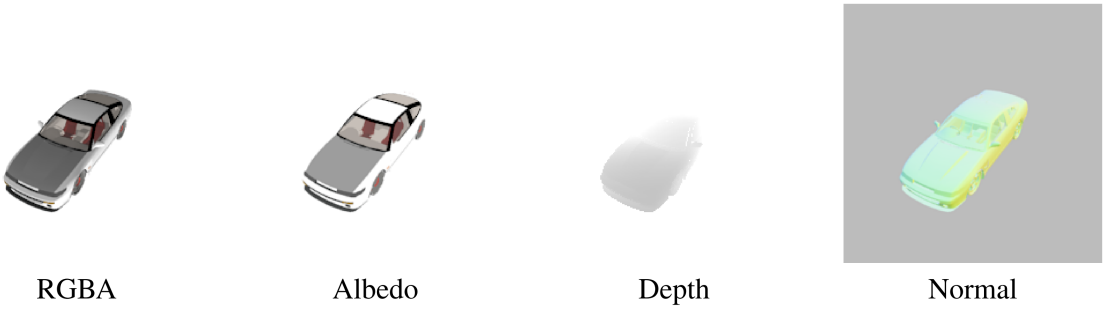
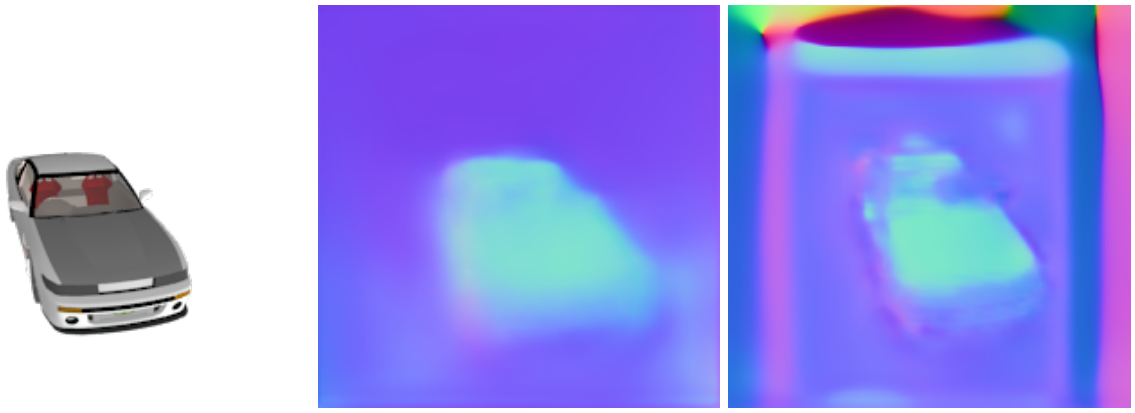


Figure 11: Example of image with maps used as ground truth taken from [26]

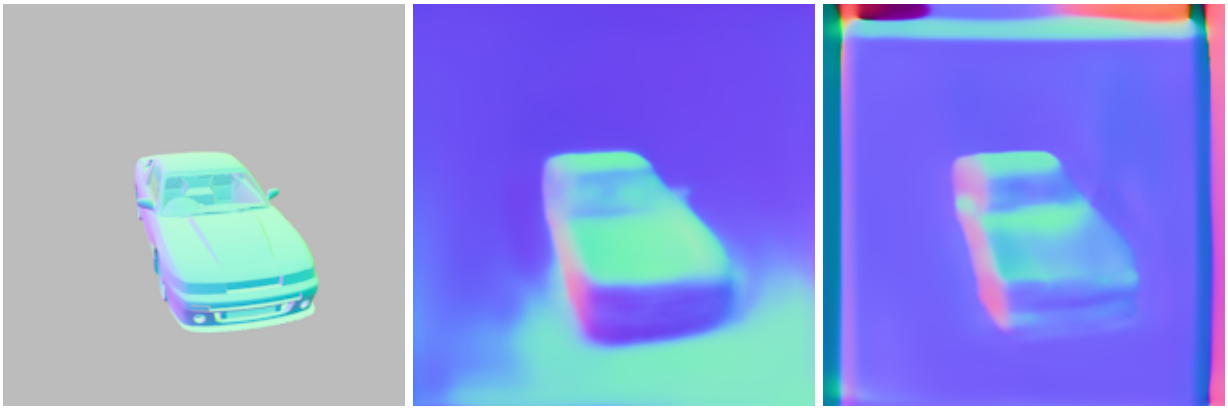
For normal map generation we used models from [26] which implements a network that estimates the per-pixel surface normal probability distribution and uses uncertainty-guided sampling to improve the quality of prediction of surface normals. The paper provided code at `https://github.com/baegwangbin/surface_normal_uncertainty` that implemented this method on a network trained on ScanNet [27], with the ground truth and data split provided by FrameNet [28], and another trained on NYUv2 [29], with the ground truth and data split provided by GeoNet [30] [31]. Both models take in the original image and dimensions of the image as input and return a corresponding normal map with the same dimensions as the given input dimensions. We run both pretrained models on the dataset.



(a) Original (b) ScanNet output (224x224) (c) NYUv2 output (224x224)

Figure 12: Comparison of original input and two model outputs

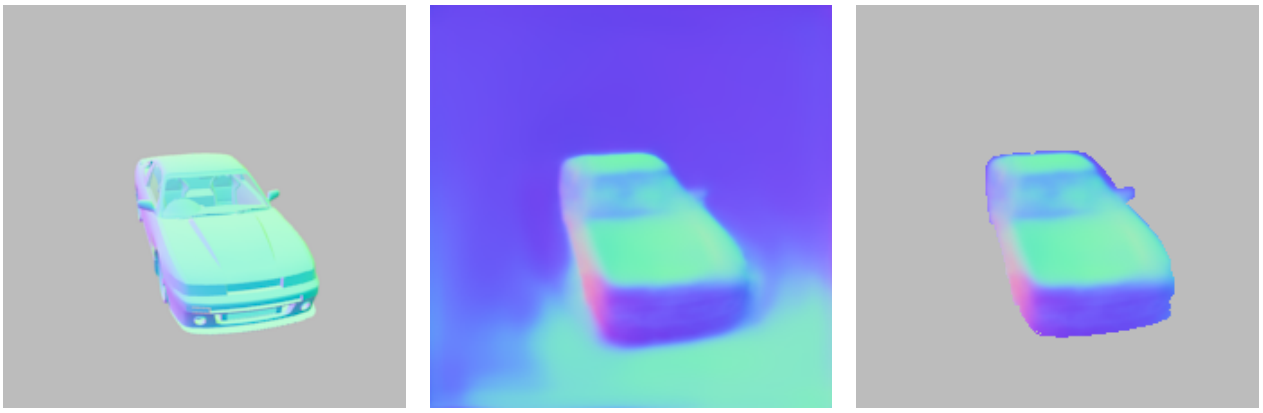
We then pass in input dimensions larger than the actual ones into the models, such that a normal map larger than the original input is produced. We then resize the image to the original input dimensions.



(a) Ground truth from [24] (b) ScanNet output (448x448) (c) NYUv2 output (448x448)

Figure 13: Comparison of model outputs when setting input dimensions as 448x448 instead of 224x224 alongside ground truth

The normal maps generated for images given larger input dimensions tend to have more clearly defined edges and surface contouring as shown in figures 13b and 13c as compared to figures 12b and 12c. It is also important to note that the ground truth for NYUv2 is only defined for the centre crop of the image and the prediction is therefore not accurate outside the centre. This is shown in figures 12c and 13c where noise is generated around the borders of the normal maps.
To compare our generated normal maps to the ground truth normal maps provided in [24], we first mask out the background of the generated normal maps such that the difference in background colour does not contribute to the evaluation metrics for normal map generation.



(a) Ground truth from [24] (b) ScanNet output (448x448) (c) Output with background masking

Figure 14: Example of masking out background for model evaluation against ground truth

We then use Pixel Based Visual Information Fidelity to compare the normal maps generated by the two models to the ground truth. Visual Information Fidelity is a reference image quality metric that quantifies the amount of visual information preserved after image processing [32] and can be used to measure various image quality attributes such as noise level and sharpness [33].

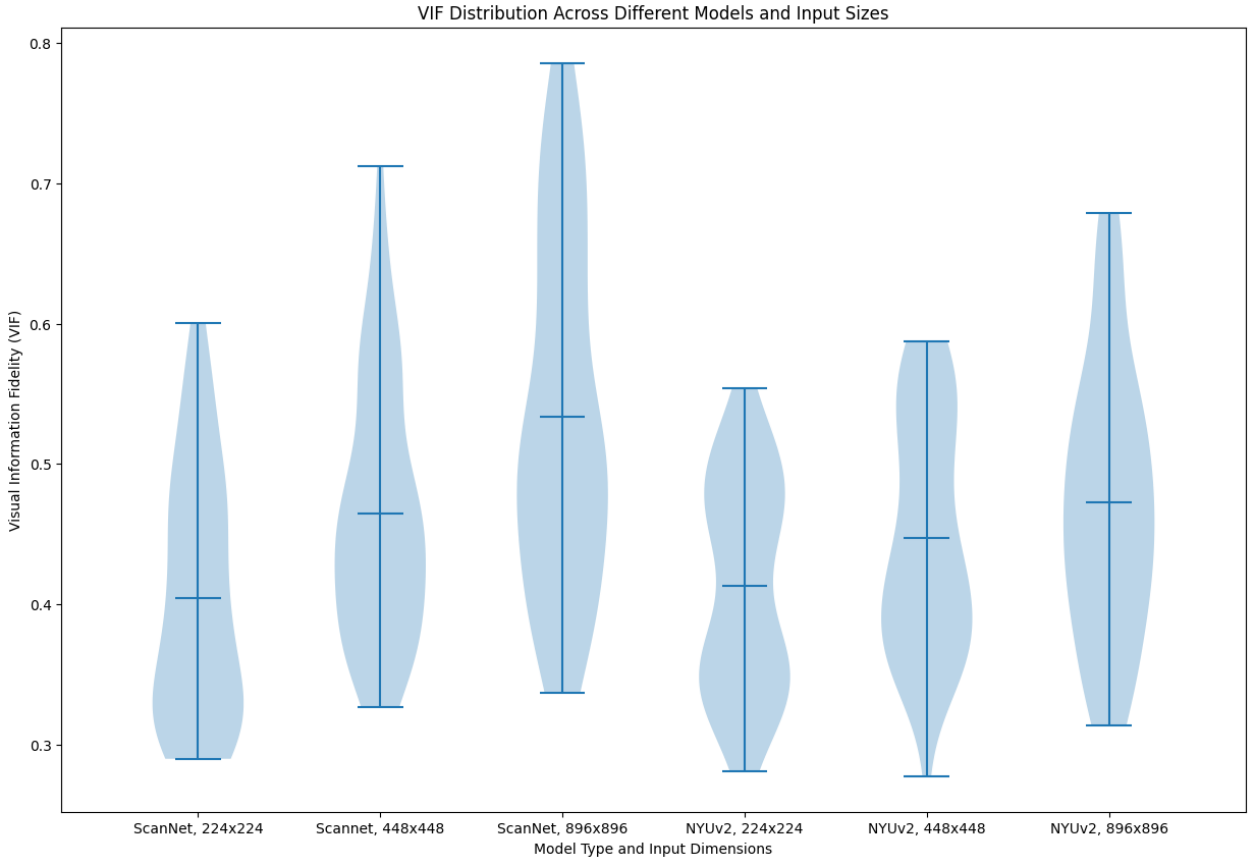


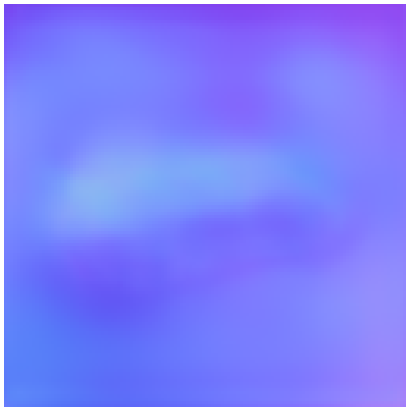
Figure 15: Comparison of VIF between ground truth and different models

Table 1: Comparison of normal map generation models on ShapeNet renders. ↑ indicates higher is better, ↓ indicates lower is better. The images are labelled easy or hard.

Difficulty	Model	Input dimensions	Mean VIF ↑	Standard deviation of VIF↓
Easy	ScanNet	224x224	0.400	0.0826
		448x448	0.463	0.0802
		896x896	**0.541**	0.119
	NYUv2	224x224	0.420	0.0736
		448x448	0.451	0.0784
		896x896	0.474	0.0901
Hard	ScanNet	224x224	0.408	0.0905
		448x448	0.466	0.0976
		896x896	**0.526**	0.125
	NYUv2	224x224	0.406	0.0749
		448x448	0.444	0.0839
		896x896	0.472	0.0981

From figure 15 and table 1 we see that normal map generation quality increases with larger input arguments and that the model trained on ScanNet on average generates normal maps that are closer to the ground truth compared to that trained on NYUv2. Hence, in the final model we decided to use the model trained on ScanNet on the ShapeNet database in [25].

(a) Original image from [25] of size 128x128



(b) Output from ScanNet model with 128x128 passed in as input dimensions

(c) Output from ScanNet model with 512x512 passed in as input dimensions

Figure 16: Original ShapeNet image and normal map outputs

Without passing in dimensions larger than the input image into the model, we can see from comparing Figures 12a and 12b to Figures 16a and 16b that the quality of the resulting normal map decreases as the resolution of the original input image decreases. Hence, we pass in larger input dimensions (512x512) during normal map generation as to achieve higher quality maps, as shown in Figure 16c.

2.3.2 Depth Map Exploration

Depth maps store the distance of a surface from the camera per-pixel. These distances vary in type, such as metric, which considers the physical distance from the camera to the observed point, and relative (such as those produced by the models below). Monocular depth estimation (MDE) models input just a singular image, and produce a depth map (relative distance).
Produced depth maps were compared against the "ground truths" within `https://github.com/Xharlie/ShapenetRender_more_variation`, provided by [24], as was done in the normal priors exploration. An

example of the depth map produced by them is visible in Figure 11. However, it is important to note that these depth map "ground truths" were not always perfect, as can be seen in the following example:

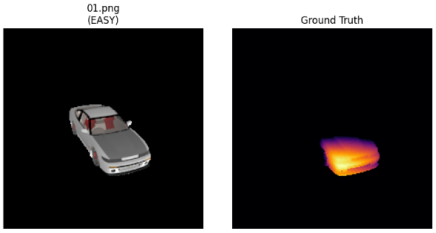


Figure 17: Example of poor depth ground truth data

This inclined us to take the quantitative results produced by comparing MDE models tested against these ground truths with a pinch of salt. For each produced depth map, the following metrics were used to compare against the ground truths.

1. **Absolute Relative Error**: Measures the average difference between the predicted depth and the ground truth, normalised by the ground truth depth.
2. **Root Mean Squared Error (RMSE)**: Calculates the standard deviation of the residual errors.
3. **Scale-invariant RMSE (SI-RMSE)**: Computes the RMSE while ignoring the unknown absolute scale and shift between the prediction and ground truth.
4. δ at **1.25** ($\delta_{1.25}$): Represents the percentage of predicted pixels p that satisfy the condition $\max(\frac{p}{p^{gt}}, \frac{p^{gt}}{p}) < 1.25$, which takes into account close pixel-wise agreement.

The following table summarises the metrics across the MiDaS models tested.

Table 2: Comparison of MiDaS models on set of easy and hard images.

Difficulty	**Model**	**AbsRel** $\downarrow$	**RMSE** $\downarrow$	**SI-RMSE** $\downarrow$	$\delta < 1.25 \uparrow$
Easy	DPT_Hybrid	0.089 ± 0.12	20.38 ± 19.39	0.123	0.909
Easy	DPT_Large	0.091 ± 0.12	20.56 ± 19.74	0.124	0.909
Easy	MiDaS_small	0.096 ± 0.13	21.54 ± 20.44	0.129	0.918
Hard	DPT_Hybrid	0.101 ± 0.15	19.65 ± 17.99	0.128	0.907
Hard	DPT_Large	0.170 ± 0.41	22.40 ± 22.12	0.151	0.906
Hard	MiDaS_small	0.190 ± 0.45	24.45 ± 23.56	0.164	0.900

The quantitative degradation of `DPT_Large` on the Hard set contradicts visual inspection. This discrepancy can be attributed to the quality of the available Ground Truth (GT) depth maps (as discussed earlier).

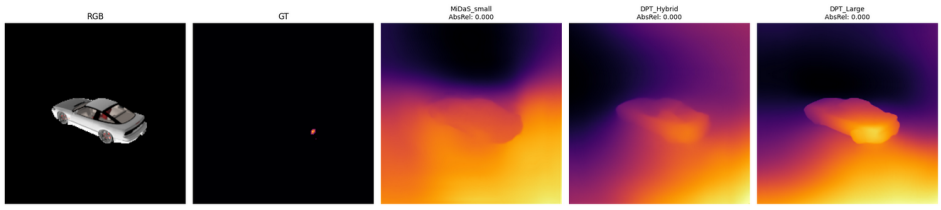


Figure 18: Depth Maps produced by MiDaS models on image with poor GT depth map.

Although `DPT_Hybrid` appears to align more closely with the GT depth map, `DPT_Large` is what is used in the final depth prior generation (as described in **2.3.4 Selected Prior Integration**). One reason is that

the model produces depth maps with cleaner edges along the object boundaries (unlike `DPT_Hybrid`), which can be seen to have closer sections of the object blend into the foreground pixels.

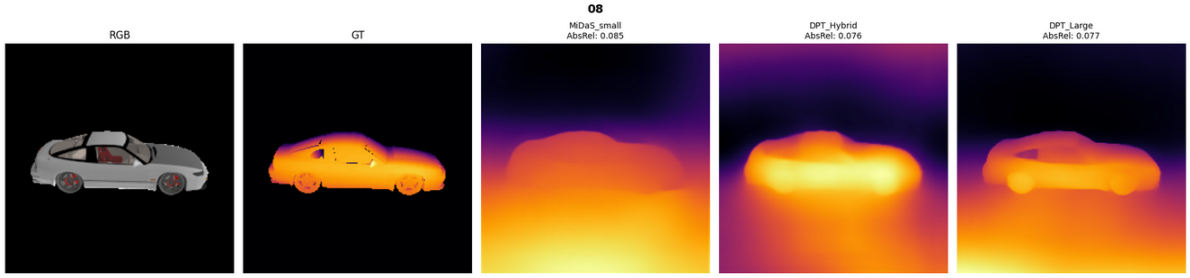


Figure 19: Depth Maps produced by the MiDaS models

We also explored the utility of explicit depth discontinuity features, or depth edges. These reflect sudden changes in depth values (when the first spatial derivative of the depth function is large) within a depth map. These can serve as a proxy for occlusion boundaries, and we were investigated as they potentially aid in separating foreground objects from background geometry.
We looked four classical edge detection operators applied directly to the estimated depth maps:

1. **Canny Edge Detector:** Tends to produce sparse, crisp boundaries, but with noisy depth maps, it can form edges that are disjoint. Available within the OpenCV (`cv2.Canny`) library.
2. **Sobel Operator:** Approximates using weighted convolution kernels, acting as a low-pass differentiator.
3. **Scharr Operator:** Similarly to Sobel, uses weights, works to improve the Sobel operator to provide better rotational symmetry.
4. **Prewitt Operator:** Similar to Sobel, but uses a simpler integer-valued kernel (uniform weighting across rows and columns).

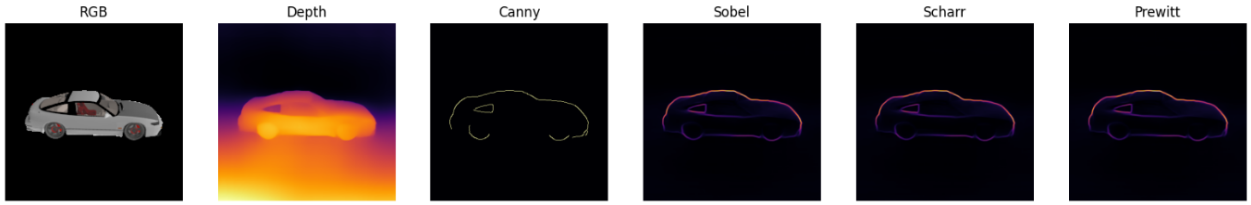


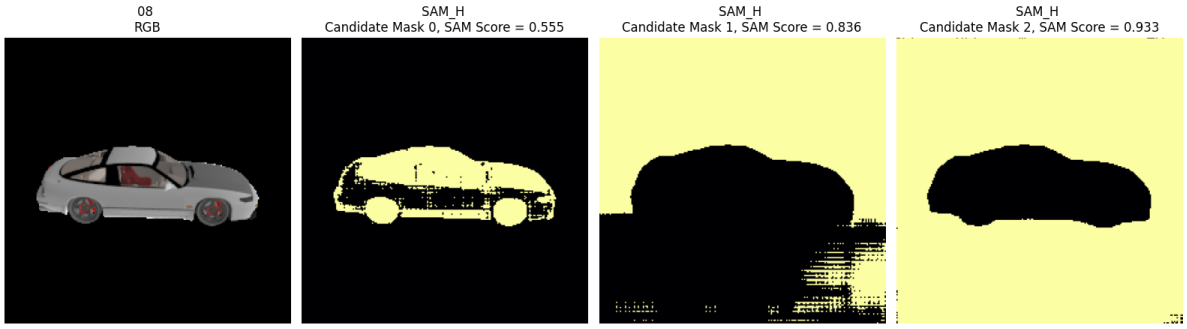
Figure 20: Edge operators applied to estimated depth maps. From left to right: Reference RGB, Estimated Depth, Canny, Sobel, Scharr, and Prewitt outputs.

Canny produces binary edges that often fragment under the noise inherent in monocular depth estimation. While these operators successfully identify major structural outlines, we found that they are at times sensitive to artifacts in the estimated depth maps. Consequently, we determined that explicit edge channels added insufficient benefits compared to other priors, and were excluded. However, while these edge operators were not used as input priors, we discuss their potential application in **Section 4.3 Future Directions**.

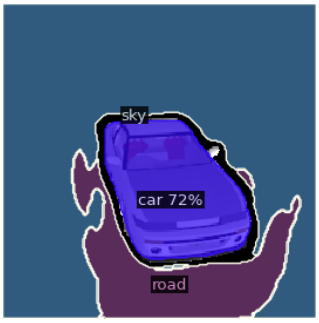
2.3.3 Segmentation and Salient Object Detection Exploration

Separating pixels belonging to the foreground object, through a segmentation mask or, as will be detailed below, using a salient object detection (SOD) model, can be another prior. This involves producing a binary mask that separates an object from its background.
Initially, we explored standard semantic and panoptic segmentation models, such as those found in the Detectron2 [34] model zoo, and the Segment Anything Model (SAM) [35]. These models are often used for

segmentation, but as illustrated in Figure 21, these produced non-contiguous masks that often had sections that included more background pixels. Segmentation models are also limited on their training classes, and despite being tested on categories in this set, their masks were improved on by salient object detection models.



(a) SAM Results (3 candidate masks produced per input)



(b) Panoptic Segmentation (Detectron2 Model) Result

Figure 21: Sample outputs from standard segmentation approaches.

SOD models identify the most visually distinct object in a scene, which allows producing a binary mask that tightly hugs the object boundary. To quantitatively evaluate SOD models, we noted that the ShapeNet images used (the same as in the normal and depth priors section) had transparent backgrounds, allowing using the alpha channel to be used as the 'ground truth' for the object silhouette.
We tested three SOD architectures: `rembg` (based on the U-2-Net architecture) [36], `InSPyReNet` [37], and `BiRefNet` [38].
Examples of the binary masks produced by the architectures, given an input image, are in figure 22

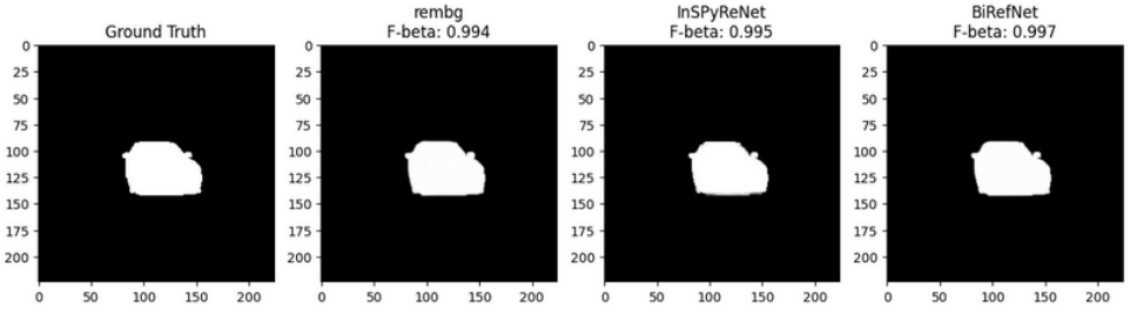


Figure 22: Example of Salient Object Detection produced binary masks

We evaluated performance using Mean Absolute Error (MAE), Intersection over Union (IoU), and F_β-

Measure. F_β is a weighted harmonic mean of precision and recall, defined as:

$$F_\beta = \frac{(1+\beta^2)\,\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}} \tag{3}$$

It is commonly used is salient object detection to assess the quality of binary masks produced. When $\beta = 1$, precision and recall are equally weighted. Greater values of β prioritise recall, while lower ones prioritise precision. We set β^2 to 0.3, following conventional practice in SOD literature, to emphasise precision over recall [39], [40], [41]. When identifying a single salient object, false positives (background pixels incorrectly classified as foreground) are often considered worse than small false negative sections along the boundary of the object.

Note that the ShapeNet images used are split into 'Easy' and 'Hard' as categories, as before.

Table 3: Comparison of Salient Object Detection models on ShapeNet renders. ↑ indicates higher is better, ↓ indicates lower is better.

Difficulty	**Model**	**IoU ↑**	F_β ↑	**MAE ↓**
Easy	rembg (U-2-Net)	**0.986**	0.991	**0.004**
	InSPyReNet	0.983	**0.996**	**0.004**
	BiRefNet	0.966	0.979	0.006
Hard	rembg (U-2-Net)	**0.980**	0.988	**0.005**
	InSPyReNet	0.966	**0.991**	0.006
	BiRefNet	0.952	0.973	0.007

Figure 23 illustrates the distribution of F_β scores across both "Easy" and "Hard" datasets. `InSPyReNet` has the tightest interquartile range, particularly on the Easy set. `rembg` demonstrates similar stability but with a slightly broader spread on the 'Hard ' dataset. `BiRefNet` is competitive (note the scale of the y-axis, with all achieving scores greater than 0.95), but comparatively shows a lower median score and higher variance, suggesting it is more sensitive to specific geometries.

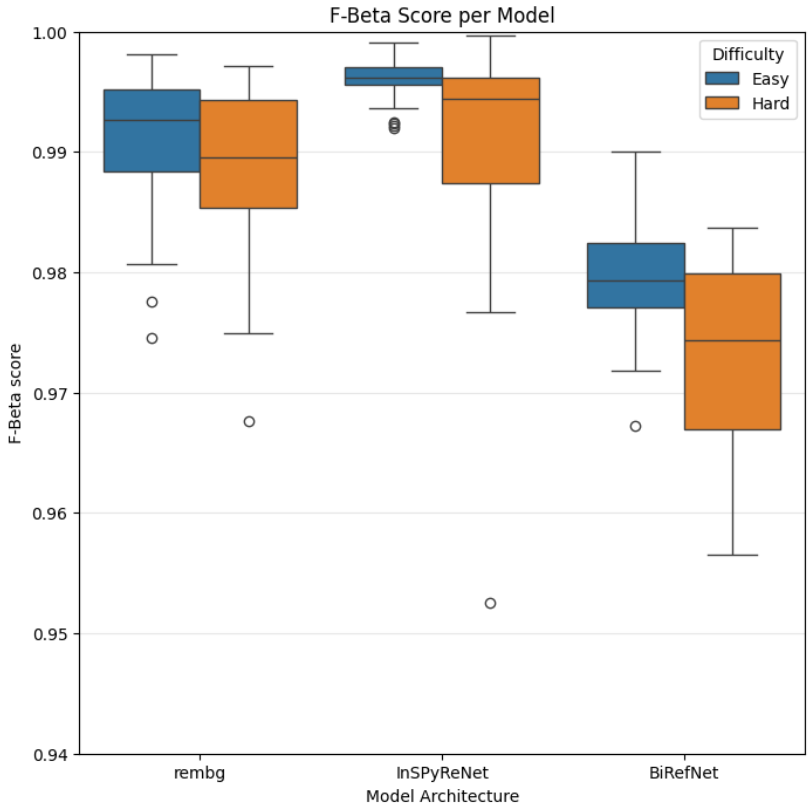


Figure 23: Distribution of F_β scores for each model across the two difficulty levels.

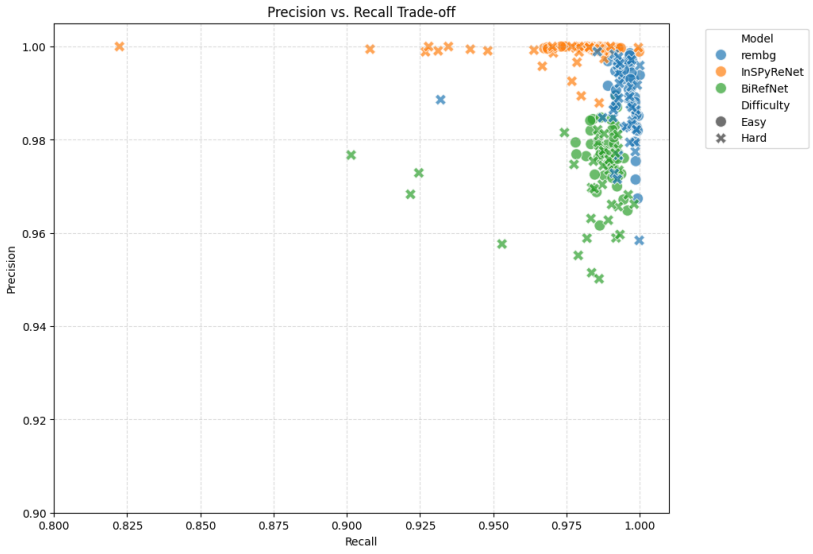


Figure 24: Precision vs. Recall trade-off

The scatter plot in Figure 24 shows how the models operate with different aims; `InSPyReNet` (in orange) clusters tightly towards the top of the high-performance region (top right), suggesting a priority of precision (often very close to 1.0). In practice, this may mean eliminating some background noise, but this may mean missing parts of the object. `rembg` (blue) also has high recall and precision, and it could also be used as a robust general-purpose model for segmentation (that does not clip parts of the object of interest).

To identify the most frequent "winner," we counted the number of images where each model achieved the highest F_β score. `InSPyReNet` achieved the highest score on the majority (28 vs `rembg`'s 8).
However, to contextualise these results, all three models achieved very high scores (with $F_\beta > 0.97$ and IOU > 0.95), largely as the ShapeNet images have a clear foreground object against a uniform background. This reduces the number of possible cases of background vs foreground confusion, which means the models differ meaningfully here on fine structures at the object boundary. In practice, these can include wing mirrors, antennae, etc. Therefore, the differences in the models come from how well they capture these fine details.
However, in the end we decided not to use SOD models to generate geometry priors for Splatter Image. This was in part due to the lack of computational resources and time but was also because we thought that since the binary mask only consisted of 1's and 0's, it may not be as expressive as normal and depth maps, which encode more information.

2.3.4 Selected Prior Integration

During model selection, basic notebooks were written to operate and evaluate the relevant models. Once a model was selected, the corresponding notebook would be refactored into a high-performance script designed to process full datasets, producing ready-processed priors for every RGB image within the given dataset. Specifically, inference code would be rewritten to ensure it operates on high performance hardware such as the GPU, and that all operations were executed in a batched manner.

```
with torch.no_grad():
    for batch_imgs, batch_filenames in loader:
        # Prediction, B C H W format
        batch_imgs = batch_imgs.to(device)
        preds = model(batch_imgs)

        # Batched downsize
        preds = torch.nn.functional.interpolate(
            preds.unsqueeze(1),
            size=target_size,
            mode="bicubic",
            align_corners=False
        ).squeeze(1)

        # Batched 0 to 1 normalization
        batch_flat = preds.flatten(start_dim=1)
        min_val, max_val = torch.aminmax(batch_flat, dim=1, keepdim=True)
        min_val = min_val.view(preds.size(0), 1, 1)
        max_val = max_val.view(preds.size(0), 1, 1)
        preds_normalized = (preds - min_val) / (max_val - min_val + 1e-8)
        preds_uint8 = preds_normalized.mul(255).byte().cpu().numpy()

        # H W format -> 128 * 128
        for j, file_id in enumerate(batch_filenames):
            full_batch.append(ProcessedImage(
                uuid=uuid,
                file_id=file_id.item(),
                image=preds_uint8[j].tobytes()
            ))
```

Listing 1: Batched processing of depths

An example of a performant rewrite is that of the depth generation python script. As shown in the excerpt, inference operates on batches supplied by a PyTorch DataLoader, with all other operations also being executed batchwise on the device (the GPU in our case); only once the batch is fully processed is it moved back to regular (host) memory, and the individual priors extracted with relevant metadata for use/saving. We also note that predicted depths are quantized to 8-bit unsigned integers, all predicted priors are quantized this way; for example normals are quantized from $3 \times$ 32-bit floats to $3 \times$ 8-bit unsigned integers; this is due to memory and storage limitations. For example the SRN cars dataset contain 387,956 images, each 128×128 pixels, storing only priors such as depths and normals as 32-bit floats for each image would require:

$$\textbf{Depths:} \quad 387,956 \times 128 \times 128 \times 4 \text{ bytes} = 25,425,084,416 \text{ bytes} \approx \mathbf{25.43} \text{ GB}$$

$$\textbf{Normals:} \quad 387,956 \times 128 \times 128 \times 3 \times 4 \text{ bytes} = 76,275,253,248 \text{ bytes} \approx \mathbf{76.28} \text{ GB}$$

This makes storing the dataset with calculated priors impractical, be it in memory or disk, quantization allows us to cut these requirements down to a quarter of the original size.

Prior generation scripts were succesfully implemented for:

- Depth
- Surface Normals
- Segmentation

All prior generation scripts can be found in the `/geometry-priors` folder within the 3DGS-priors repository.

As to improve training and evaluation performance, we chose to generate priors in advance for selected datasets. As such we developed a pipeline that executes the prior generation scripts and constructs a ready-to-use dataset, alongside a custom Torch Dataset class that can read said dataset. Implementation details can be found in **Section 2.4: Data pipelines**.

2.3.5 Model Changes

The first change to the model was to have the top-level `GaussianSplatPredictor` class dynamically calculate the number of input channels required based on the training configuration, this information was then passed to the underlying Convolutional layers and UNet blocks during initialisation.

```python
def calc_channels(cfg):
    # Base RGB channels
    in_channels = 3

    # Older configs may not have relvant options, select() returns None if the option is missing
    if OmegaConf.select(cfg, "data.use_pred_depth") is True:
        in_channels += 1
    if OmegaConf.select(cfg, "data.use_pred_normal") is True:
        in_channels += 3

    return in_channels
```

Listing 2: Channel calculation code

```python
# Calculate number of input channels
self.in_channels = calc_channels(cfg)

# Initialise correct model depending on if Gaussian mean offsets are to be calculated
if cfg.model.network_with_offset:
    split_dimensions, scale_inits, bias_inits = self.get_splits_and_inits(True, cfg)
    self.network_with_offset = networkCallBack(cfg,
                                cfg.model.name,
                                self.in_channels,
                                split_dimensions,
                                scale = scale_inits,
                                bias = bias_inits)
```

```
    assert not cfg.model.network_without_offset, "Can only have one network"
if cfg.model.network_without_offset:
    split_dimensions, scale_inits, bias_inits = self.get_splits_and_inits(False, cfg)
    self.network_wo_offset = networkCallBack(cfg,
                                cfg.model.name,
                                split_dimensions,
                                scale = scale_inits,
                                bias = bias_inits)
    assert not cfg.model.network_with_offset, "Can only have one network"
```

Listing 3: New model initialisation code

The SongUNet that Splatter Image is built on already implements support for FiLM layers in its UNet blocks, using a 2 layer FCN as the FiLM parameter generator. Splatter Image's top level `GaussianSplatPredictor` only allows FiLM to be used to inject camera pose embeddings. We provide a new configuration option, `custom_embedding`, and modify GaussianSplatPredictor's forward pass, allowing a custom embedding to be passed in the forward pass that overrides the camera pose embedding (as camera pose embeddings are often not used in the project's datasets, such as `SRN cars`), allowing the embedding to be used by the model without any architectural changes.

```
def forward(self, x,
            source_cameras_view_to_world,
            source_cv2wT_quat=None,
            focals_pixels=None,
            activate_output=True,
            custom_emb = None):

    ...

    # Get embedding for modulation
    film_emb = None
    if self.cfg.cam_embd.embedding is not None:
        if "custom_embedding" in self.cfg.data and self.cfg.data.custom_embedding:
            assert custom_emb is not None
            film_emb = custom_emb
        else:
            # Get camera embedding
            cam_embedding = self.get_camera_embeddings(source_cameras_view_to_world)
            assert self.cfg.cam_embd.method == "film"
            film_emb = cam_embedding.reshape(B*N_views, cam_embedding.shape[2])

    ...

    # Runs input through SingleImageSongUNetPredictor
    if self.cfg.model.network_with_offset:
        split_network_outputs = self.network_with_offset(x,
                                                         film_camera_emb=film_emb,
                                                         N_views_xa=N_views_xa
                                                         )
```

Listing 4: New GaussianSplatPredictor forward pass

The selected segmentation model did not produce classification or instance labels/tokens, as such we did not test this FiLM implementation with multimodal inputs, leaving that for future investigation.

One of our desired features was to allow a model with new priors to be fine-tuned using existing weights, namely those from the pretrained Splatter Image models. This was achieved by grafting the old weights of a pretrained model onto a new instance of a GaussianSplatPredictor. This grafting mechanism was implemented externally to the model, as such its implementation details are covered in **Section 2.3.7: Training and Evaluation Changes**.

2.3.6 Manual LoRA Integration

To adapt Splatter Image for fine-tuning on the ShapeNet-SRN Cars dataset (with depth and normal priors), we added Low-Rank Adaptation (LoRA) [21] into the GaussianPredictor module of Splatter Image. LoRA adds low-rank trainable matrices into frozen, pretrained layers. This preserves the base model, while further fine-tuning it to the new data.

Existing libraries such as Hugging Face's PEFT [42] provide out-of-the-box LoRA integration, we found them incompatible with the weight-grafting mechanism required by our SplatterImage training pipeline. Due to compute limitations, we rely on Splatter Image grafting weights from pretrained checkpoints where input channel dimensions change (e.g., when adding depth or normal map channels). This leads to shape mismatches during initialisation steps (such as for PEFT's LoRA layers) since the pre-trained weights cannot be directly loaded into layers with modified input dimensions.

We added LoRA manually within the `GaussianPredictor` and `train_network` modules, using code from Microsoft's `loralib` [22]. We modified the underlying `Linear` and `Conv2d` layers of the U-Net architecture to include a `LoRALayer` mixin. This allows the freezing of pre-trained base weights while injecting the LoRA matrices (A and B) directly into the forward pass. As mentioned earlier, this preserves the frozen pretrained weights from the original 3-channel RGB model, but is compatible with the new channels.

The `LoRALayer` mixin is integrated into both `Linear` and `Conv2d` classes within the U-Net architecture. Each layer checks during initialization whether `lora_rank > 0`, and if so, creates the low-rank matrices alongside the frozen base weights:

```
if lora_rank > 0 and kernel > 0 and self.weight is not None and not self.up and not self.down:
    # LoRA A: [r, in * k * k], LoRA B: [out, r]
    self.lora_A = nn.Parameter(self.weight.new_zeros((self.r, in_channels * kernel * kernel)))
    self.lora_B = nn.Parameter(self.weight.new_zeros((out_channels, self.r)))
    self.scaling = float(lora_alpha) / max(1, float(self.r))
    self.weight.requires_grad = False
    nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
    nn.init.zeros_(self.lora_B)
```

Listing 5: LoRA Parameter Creation in Conv2d

These LoRA parameters are distributed throughout the U-Net via the `block_kwargs` dictionary in `SongUNet.__init__()`. The rank, alpha, and dropout values are passed to each `UNetBlock`, which then forwards them to its constituent `Conv2d` and `Linear` layers.

We have `merge_lora_weights()` and `unmerge_lora_weights()` functions to fuse or separate LoRA adaptations from base weights. During merging, the LoRA delta $\frac{\alpha}{r}BA$ is computed and added directly to the frozen weights, eliminating the two-branch computation overhead during inference.

```
def merge_lora_weights(model: nn.Module):
    for m in model.modules():
        if isinstance(m, (Linear, Conv2d)) and m.has_lora and not getattr(m, "_lora_merged", False):
            with torch.no_grad():
                if isinstance(m, Linear):
                    # delta = B @ A -> shape (out, in)
                    delta = (m.lora_B @ m.lora_A).to(m.weight.dtype) * m.scaling
                    m.weight.data += delta
                    m._lora_merged = True
                else:  # Conv2d
                    delta = (m.lora_B @ m.lora_A).view_as(m.weight) * m.scaling
                    m.weight.data += delta
                    m._lora_merged = True
```

Listing 6: LoRA Weight Merging (from gaussian_predictor_lora.py)

This merging is standard practice in LoRA implementations [22] and is equivalent to the two-branch forward pass, while being more computationally efficient.

2.3.7 Training and Evaluation Changes

The existing Splatter Image evaluation script needed minimal changes, thanks to how we integrated our priors into the project. The training script initially needed minimal changes, but then was split into two versions, one with minimal changes for standard training, and one with additional changes required for LoRA support.

For evaluation code, the first change was adding support for loading our pretrained models available on HuggingFace.

```
# Load pretrained model from HuggingFace if no local model specified
if args.experiment_path is None:
    # Eval run on the our new dataset with priors
    if dataset_name in ["cars_priors"]:
        cfg_path = hf_hub_download(repo_id="MVP-Group-Project/splatter-image-priors",
                                   filename="model-depth-normal/config.yaml")
        model_path = hf_hub_download(repo_id="MVP-Group-Project/splatter-image-priors",
                                filename="model-depth-normal/model_best.pth")

    # Eval run on previous Splatter Image datasets
    else:
        cfg_path = hf_hub_download(repo_id="szymanowiczs/splatter-image-v1",
                                   filename="config_{}.yaml".format(dataset_name))
        if dataset_name in ["gso", "objaverse"]:
            model_name = "latest"
        else:
            model_name = dataset_name
        model_path = hf_hub_download(repo_id="szymanowiczs/splatter-image-v1",
                                filename="model_{}.pth".format(model_name))

else:
    cfg_path = os.path.join(experiment_path, ".hydra", "config.yaml")
    model_path = os.path.join(experiment_path, "model_latest.pth")

# load cfg
training_cfg = OmegaConf.load(cfg_path)
```

Listing 7: New model loading code

Input preparation was also changed, namely priors are concatenated (if enabled) as additional channels to the input RGB images, before being fed into the network. Splatter Image uses the PyTorch DataLoader alongside custom Dataset classes to load batches of RGB images to perform inference on. The Dataset classes do not directly return a tensor of images, but a dictionary containing relevant batch data:

```
images_and_camera_poses = {
    "gt_images": self.all_rgbs[example_id][frame_idxs].clone(),
    "world_view_transforms": self.all_world_view_transforms[example_id][frame_idxs],
    "view_to_world_transforms": self.all_view_to_world_transforms[example_id][frame_idxs],
    "full_proj_transforms": self.all_full_proj_transforms[example_id][frame_idxs],
    "camera_centers": self.all_camera_centers[example_id][frame_idxs]
}

images_and_camera_poses = self.make_poses_relative_to_first(images_and_camera_poses)
images_and_camera_poses["source_cv2wT_quat"] = self.get_source_cw2wT(images_and_camera_poses
    ["view_to_world_transforms"])

return images_and_camera_poses
```

Listing 8: Excerpt of srn.py Dataset code

This allowed our priors to be introduced into the evaluation code with ease, by simply creating a new custom Torch Dataset class in `srn_priors.py` that provides priors as new key/value pairs in the returned dictionary. Priors were specifically returned as PyTorch tensors matching the RGB image batch. The priors can then be accessed and concatenated with the RGB images in a specific order to generate the final input tensor for Splatter Image to perform inference on. Assertions are also performed to verify that priors are indeed available for the currently active Dataset.

```
# Concatenate selected priors
input_images = data["gt_images"][:, :model_cfg.data.input_images, ...]
if "use_pred_depth" in model_cfg.data and model_cfg.data.use_pred_depth:
    assert model_cfg.data.category == "cars_priors", "Dataset does not have predicted maps!"
    input_images = torch.cat([input_images,
                    data["pred_depths"][:, :model_cfg.data.input_images, ...]],
                    dim=2)
if "use_pred_normal" in model_cfg.data and model_cfg.data.use_pred_normal:
    assert model_cfg.data.category == "cars_priors", "Dataset does not have predicted maps!"
    input_images = torch.cat([input_images,
                    data["pred_normals"][:, :model_cfg.data.input_images, ...]],
                    dim=2)

# Get camera to center depth
if model_cfg.data.origin_distances:
    input_images = torch.cat([input_images,
                    data["origin_distances"][:, :model_cfg.data.input_images, ...]],
                    dim=2)
```

Listing 9: Splatter Image input tensor preparation code

For the training code, two changes had to be made. The first change was to model loading, adding support for training using existing HuggingFace model weights from the base Splatter Image project.

```
# Resume from HuggingFace pretrained weights, perform weight graft if now training with
    additional priors
elif cfg.opt.pretrained_hf:
    category = cfg.data.category
    if category == "cars_priors":
        category = "cars"

    model_name = category
    if cfg.data.category in ["gso", "objaverse"]:
        model_name = "latest"

    cfg_path = hf_hub_download(repo_id="szymanowiczs/splatter-image-v1", filename="config_
    {}.yaml".format(category))
    model_path = hf_hub_download(repo_id="szymanowiczs/splatter-image-v1", filename="model_
    {}.pth".format(model_name))
    old_cfg = OmegaConf.load(cfg_path)
    assert is_base_model(old_cfg)

    checkpoint = torch.load(model_path, map_location=device, weights_only=False)

    # Check if new model uses priors
    if is_base_model(cfg):
        try:
            gaussian_predictor.load_state_dict(checkpoint["model_state_dict"])
        except RuntimeError:
            gaussian_predictor.load_state_dict(checkpoint["model_state_dict"], strict=False)
    else:
        gaussian_predictor = graft_weights_with_channel_expansion(checkpoint["
    model_state_dict"], gaussian_predictor, old_cfg, cfg)
        print("Grafting performed successfully")

    best_PSNR = checkpoint["best_PSNR"]
    print('Loaded model from a pretrained Huggingface checkpoint')
```

```
    OmegaConf.save(config=cfg, f=os.path.join(vis_dir, "config.yaml"))
```

Listing 10: New training setup code

The program first downloads the appropriate base splatter image configuration and model files. The current training configuration is then checked to see if it uses any priors, with `is_base_model` returning true if no priors are to be used. In the case priors have been requested, the new gaussian_predictor instance will have mismatched layer dimensions compared to the pretrained Splatter Image model, meaning the pretrained weights cannot be directly loaded into gaussian_predictor, in this case weights are copied manually into a state dictionary with layers of correct dimensions, we refer to this process as grafting, and is achieved using the `graft_weights_with_channel_expansion` function in `prior_utils.py`.

```
def graft_weights_with_channel_expansion(old_state_dict, new_model, old_cfg, new_cfg):
    new_state_dict = new_model.state_dict()

    # Iterate over all layers
    for name, new_param in new_state_dict.items():
        if name not in old_state_dict:
            # New LoRA parameters not in base model checkpoint. Skip to avoid KeyError.
            if "lora_" in name:
                continue
            print("Failed to find layer {} in HuggingFace model state_dict".format(name))
            raise Exception("Mismatched source model for graft")

        old_param = old_state_dict[name]

        # Directly copy tensors if matching in size (handles most layers)
        if (new_param.shape == old_param.shape):
            new_state_dict[name] = old_param.clone()
            continue

        # In theory we should only reach here for Conv2D layers, as such only need to handle weights, and these should only have extra channels in shape[1]
        if ('weight' in name):
            # Dimension check for Conv2D weights
            if new_param.dim() == 4 and old_param.dim() == 4:
                assert new_param.shape[0] == old_param.shape[0], "Grafting only supported for adding channels, not changing resolution"
                assert new_param.shape[1] > old_param.shape[1], "Cannot truncate channels during graft, can only add channels"

                new_weights = new_param.clone()
                new_weights[:, :old_param.shape[1], :, :] = old_param
                new_weights[:, old_param.shape[1]:, :, :] = 0.0

                new_state_dict[name] = new_weights
            else:
                 print(f"Warning: Skipping graft for {name} due to dimension mismatch")
        else:
            raise Exception("Failed layer graft")

    new_model.load_state_dict(new_state_dict)
    return new_model
```

Listing 11: Grafting code

Grafting works by iterating over the newly configured model's state dictionary, looking up the equivalent layers in the pretrained model's state dictionary. If equivalent layers match in shape, that means no changes have been made and the tensor from the pretrained model's layer is copied directly into the new state dictionary. If there is a mismatch in shape however, we create a new tensor matching the new model layer's shape and manually copy the values from the pretrained layer into the appropriate lower dimensions of the tensor, then zero-initialize all remaining elements in the tensor. By zero-initializing the newly added parameters, the expanded layer remains mathematically equivalent to the original. This ensures the new model's

output is identical to that of the pretrained one, despite the change in architecture. The newly configured model is reloaded with the newly generated state dictionary and finally the ready model is returned. This grafting mechanism works despite the behaviour of the newly configured model remaining the same, as back-propagation will compute non-zero gradients for the new inputs channels, allowing the model to update the zero-initialized parameters and gradually integrate the new channels, making them useful.

The second change to training code was similar to that of the evaluation script, updating input preparation, the resulting updated code block is identical to that in **Listing 6**.

For the LoRA-specific training script, within the modified `GaussianPredictor`, `requires_grad=False` is explicitly set for all pre-trained backbone weights (see `gaussian_predictor_lora.py`), leaving only the low-rank matrices A and B as trainable. This ensures that optimizer updates are restricted to the adapter layers, with instances defined as `LoRALayer`.

For the LoRA-specific training script (`train_network_lora.py`), parameter freezing is enforced differently. After model initialization and pretrained weight loading (which may include grafting), only LoRA parameters are set to trainable:

```python
if cfg.opt.lora_finetune:
    # LoRA finetuning, only finetuning adapters
    for name, p in gaussian_predictor.named_parameters():
        p.requires_grad = ("lora" in name.lower())
else:
    # Full finetune
    for p in gaussian_predictor.parameters():
        p.requires_grad = True

trainable = [p for p in gaussian_predictor.parameters() if p.requires_grad]
optimizer = torch.optim.Adam(
    trainable,
    lr=cfg.opt.base_lr,
    eps=1e-15,
    betas=cfg.opt.betas
)
```

Listing 12: LoRA Parameter Freezing (from train_network_lora.py)

This selective freezing ensures the optimizer only updates `lora_A` and `lora_B` matrices. LoRA configurations resulted in a small percent of the original parameters being trainable (compared with Splatter Image). The dedicated evaluation script `eval_lora.py` handles LoRA weights automatically, with a merging step standard in LoRA evaluation [22].

2.4 Data pipelines

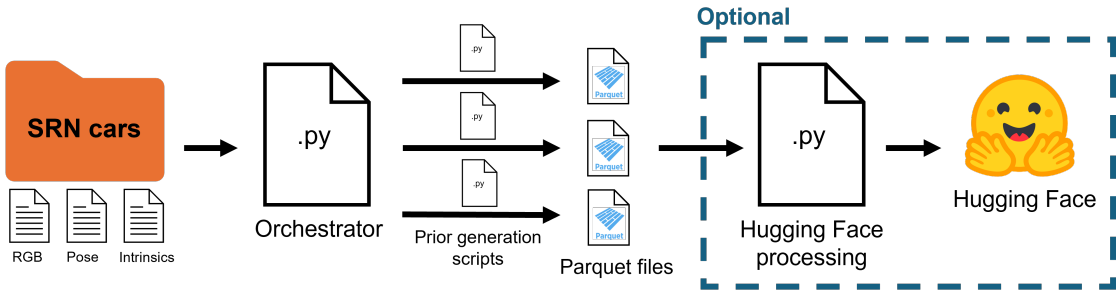


Figure 25: Data Pipeline Diagram

For training and evaluation performance reasons, we chose to generate priors in advance for selected datasets. As such we developed a pipeline that follows an orchestrator pattern; a central notebook initializes a Virtual Machine (VM) for each prior generation model, then installs the necessary dependencies into them, defined in the relevant `PriorName_requirements.txt` files located in the `/geometry-priors` folder. Once the environment is configured, the orchestrator executes the prior generation scripts to produce a complete, ready-to-use dataset.

```
# Create venv for each prior model
!python3 -m venv /content/models/depth --without-pip
!python3 -m venv /content/models/normal --without-pip

# Have to manually install pip to correctly build venvs on colab
!curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
!/content/models/depth/bin/python3 get-pip.py
!/content/models/normal/bin/python3 get-pip.py

# Verify venv's work
!/content/models/depth/bin/pip --version
!/content/models/normal/bin/pip --version

!ls -l /content/models/
```

Listing 13: VM setup

```
# Setup models

# Depth
!/content/models/depth/bin/pip install -r /content/3dgs-priors/geometry-priors/
    depth_requirements.txt

# Normals
!git clone https://github.com/baegwangbin/surface_normal_uncertainty.git /content/3dgs-
    priors/geometry-priors/surface_normal_uncertainty/
!mkdir /content/3dgs-priors/geometry-priors/surface_normal_uncertainty/checkpoints/
!gdown 1lOgY9sbMRW73qNdJze9bPkM2cmfA8Re- -O /content/3dgs-priors/geometry-priors/
    surface_normal_uncertainty/checkpoints/scannet.pt
!/content/models/normal/bin/pip install -r /content/3dgs-priors/geometry-priors/
    normal_requirements.txt
```

Listing 14: Package setup

```
# Launch models to process dataset
!/content/models/normal/bin/python /content/3dgs-priors/geometry-priors/generate_normal.py
    --in_folder="{in_folder}" --out_folder="{out_folder}" --save_iter=250

!/content/models/depth/bin/python /content/3dgs-priors/geometry-priors/generate_depth.py --
    in_folder="{in_folder}" --out_folder="{out_folder}" --save_iter=250

!python /content/3dgs-priors/geometry-priors/generate_base.py --in_folder="{in_folder}" --
    out_folder="{out_folder}" --save_iter=500
```

Listing 15: Dataset and prior generation script execution

The processing scripts generate `parquet` format files containing the dataset information alongside relevant metadata. Parquet files are chosen as they are the industry standard for storing datasets, as the file format allows for efficient per column compression, support for streaming and quick read times. Currently, 5 parquet files are generated by `generate_cars_datasets.ipynb`, an example implementation of this pipeline for ShapeNet cars specifically, their contents are:

srn_cars_intrins UUID, Model Intrinsics

srn_cars_poses UUID, Frame ID, Pose matrix

srn_cars_rgbs UUID, Frame ID, RGB Image

srn_cars_depths UUID, Frame ID, Depth Image

srn_cars_normals UUID, Frame ID, Normal Image

These files provide a ready-to-use dataset. If the dataset is to be uploaded to HuggingFace, additional processing must be performed to transform the files into a file structure that's recognisable by HF for purposes of correctly indexing splits and subsets. Specifically, we transform the dataset to be compatible with this style of HuggingFace dataset YAML configuration:

```yaml
---
configs:
- config_name: srn_cars_intrins
  data_files:
  - split: train
    path: intrins/train/*.parquet
  - split: test
    path: intrins/test/*.parquet
  - split: val
    path: intrins/val/*.parquet

- config_name: srn_cars_poses
  data_files:
  - split: train
    path: poses/train/*.parquet
  - split: test
    path: poses/test/*.parquet
  - split: val
    path: poses/val/*.parquet

- config_name: srn_cars_rgbs
  data_files:
  - split: train
    path: rgbs/train/*.parquet
  - split: test
    path: rgbs/test/*.parquet
  - split: val
    path: rgbs/val/*.parquet

- config_name: srn_cars_depths
  data_files:
  - split: train
    path: depths/train/*.parquet
  - split: test
    path: depths/test/*.parquet
  - split: val
    path: depths/val/*.parquet

- config_name: srn_cars_normals
  data_files:
  - split: train
    path: normals/train/*.parquet
  - split: test
    path: normals/test/*.parquet
  - split: val
    path: normals/val/*.parquet
---
```

Listing 16: srn_cars_priors HuggingFace file structure config

```
# Split generated parquets into correct folder/file structure for HuggingFace upload
```

```
%cd /content/
!mkdir upload
%cd upload

datatypes = ["rgbs", "intrins", "poses", "depths", "normals"]
splits = ["train", "test", "val"]

for datatype in datatypes:
    # Make a directory for the current dataset file
    !mkdir {datatype}
    %cd {datatype}
    load_path = os.path.join(out_folder, "srn_cars_" + datatype + ".parquet")
    dataset = load_dataset("parquet", data_files=load_path)['train']
    for split in splits:
        # Generate split specific directory
        !mkdir {split}
        save_path = "/content/upload/{}/{}/{}.parquet".format(datatype, split, datatype)
        filter_split = lambda data: data['split'] == split
        filtered = dataset.filter(filter_split).to_pandas().drop(columns=['split'])
        filtered.to_parquet(save_path)
    %cd ..
```

Listing 17: HuggingFace upload preparation code

A diagram depicting this pipeline can be seen at Figure 25.

We offer one ready dataset computed using this pipeline on HuggingFace at `https://huggingface.co/datasets/MVP-Group-Project/srn_cars_priors`. More details about ShapeNet cars, which the precomputed dataset is based off, can be found in **Section 3.1: Datasets**.

Training and evaluation scripts using PyTorch DataLoaders to load datasets into memory for inference. To allow easy use of our precomputed dataset, we implement a custom PyTorch Dataset class, `srn_priors.py`, which can be plugged into DataLoaders without any additional code changes. Our custom Dataset class first downloads the datasets from Huggingface and then preprocesses for later use. Namely, the datasets are converted from Huggingface dataset objects to pandas dataframes, where rows are also grouped by model UUID and the groups saved to a dictionary, allowing efficient model to RGB image lookup. The HuggingFace datasets and redundant pandas dataframes are deallocated from memory as soon as rows are processed, as to minimize memory usage by the precomputed dataset.

```
# Download ready dataset from HuggingFace
print("Started downloading datasets")
dataset_intrins = load_dataset(
    "MVP-Group-Project/srn_cars_priors",
    name="srn_cars_intrins",
    split=self.dataset_name
)
dataset_poses = load_dataset(
    "MVP-Group-Project/srn_cars_priors",
    name="srn_cars_poses",
    split=self.dataset_name
)
dataset_rgbs = load_dataset(
    "MVP-Group-Project/srn_cars_priors",
    name="srn_cars_rgbs",
    split=self.dataset_name
)
dataset_depths = load_dataset(
    "MVP-Group-Project/srn_cars_priors",
    name="srn_cars_depths",
    split=self.dataset_name
)
dataset_normals = load_dataset(
    "MVP-Group-Project/srn_cars_priors",
```

```
    name="srn_cars_normals",
    split=self.dataset_name
)
print("Downloaded datasets")

# Convert intrinsics files to a dataframe for performance reasons
pre_dataset_intrins = dataset_intrins.to_pandas()
pre_dataset_intrins.sort_values(by=["uuid"], ascending=[True], inplace=True)

# Calculate dataset length
assert len(dataset_poses) == len(dataset_rgbs)
if cfg.data.subset != -1:
    assert cfg.data.subset > 0
    assert len(pre_dataset_intrins) >= cfg.data.subset
    self.subset_length = cfg.data.subset
else:
    self.subset_length = len(pre_dataset_intrins)

self.dataset_intrins = pre_dataset_intrins.iloc[:self.subset_length]
uuids = set(self.dataset_intrins['uuid'].unique())

# Convert remaining HF datasets to dataframes indexed by uuid
self.dataset_poses = process_and_chunk(dataset_poses, uuids)
print("Converted poses")
self.dataset_rgbs = process_and_chunk(dataset_rgbs, uuids)
print("Converted rgbs")
self.dataset_depths = process_and_chunk(dataset_depths, uuids)
print("Converted depths")
self.dataset_normals = process_and_chunk(dataset_normals, uuids)
print("Converted normals")

print("Dataset intrin length: {}".format(self.subset_length))
```

Listing 18: Excerpt from custom PyTorch Dataset

During and after training, the model is checkpointed and saved to disk. The frequency of mid-training checkpoints is set by the training configuration. Namely, the files saved are:

config.yaml Training run configuration.

model_best.pth Weights, PSNR and iteration count of the highest scoring model from training run.

model_latest.pth Most recent weights, PSNR and iteration count.

scores.txt Scores from evaluation of the model during training.

train_network.log Log of critical events during training run.

`training.ipynb` also offers code that can be directly upload these checkpoints to HuggingFace into an appropriately named folder based on the training configuration. Pretrained models for the precomputed ShapeNet `cars_priors` dataset can be found on HuggingFace at: `https://huggingface.co/MVP-Group-Project/splatter-image-priors`

```
# Save checkpoint to huggingface

files = []
model_path = Path(model_path)

# If model data is directly in folder, use that, otherwise we assume the wandb file
    structure
possible_files = [file for file in model_path.glob('*{}'.format(".pth"))]
if len(possible_files) > 0:
    files = [file for file in model_path.iterdir() if file.is_file()]
else:
    # Get top level yyyy-mm-dd folders
```

```
    date_folders = sorted([folder for folder in model_path.iterdir() if folder.is_dir()])

    # Verify some training run was performed
    if not date_folders:
        print("No date folders found.")
    else:
        latest_date_folder = date_folders[-1]
        print("Latest Date: {}".format(latest_date_folder.name))

        # Get hh-mm-ss folders
        time_folders = sorted([folder for folder in latest_date_folder.iterdir() if folder.
    is_dir()])
        if not time_folders:
            print("No time folders found inside {}".format(latest_date_folder.name))
        else:
            latest_time_folder = time_folders[-1]
            print(f"Latest Time: {latest_time_folder.name}")

            # Get files in most recent folder combination
            files = [p for p in latest_time_folder.iterdir() if p.is_file()]

login(token=hf_token)
api = HfApi()

# Upload model
for file_path in files:
    # Title upload folder correctly
    path_in_repo = "model"
    if use_depth:
        path_in_repo += "-depth"
    if use_normal:
        path_in_repo += "-normal"
    if use_lora:
        path_in_repo += "-finetune"

    path_in_repo += "/" + os.path.basename(file_path)

    # Upload file
    api.upload_file(
        path_or_fileobj=file_path,
        repo_id=repo_id,
        repo_type="model",
        path_in_repo=path_in_repo,
    )
```

Listing 19: HuggingFace automated model upload code

2.5 Training Procedures

The Splatter Image model in fully implemented using PyTorch. We use the existing training loop implementation. The `train.py` script implements the training loop using PyTorch DataLoader and Datasets classes to load in dataset entries in a batched manner. Wandb is used to perform logging, track training runs, configurations and track the model performance. Hyrda alongside Omegaconf is used to supply a training configuration to the training loop and model.

```
@hydra.main(version_base=None, config_path='configs', config_name="default_config")
def main(cfg: DictConfig):
...
```

Listing 20: Initialisation excerpt from training code

```yaml
---
defaults:
  - wandb: defaults
  - hydra: defaults
  - cam_embd: defaults
  - _self_
general:
  device: 0
  random_seed: 0
  num_devices: 1
  mixed_precision: false
data:
  training_resolution: 128
  subset: -1
  input_images: 1
  origin_distances: false
  use_pred_depth : false
  use_pred_normal : false
  custom_embedding : false
opt:
  iterations: 800001
  base_lr: 0.00005
  batch_size: 8
  betas:
    - 0.9
    - 0.999
  loss: l2
  imgs_per_obj: 4
  ema:
    use: true
    update_every: 10
    update_after_step: 100
    beta: 0.9999
  lambda_lpips: 0.0
  pretrained_ckpt: null
  pretrained_hf: False
  lora_finetune : false

model:
  max_sh_degree: 1
  inverted_x: false
  inverted_y: true
  name: SingleUNet
  opacity_scale: 1.0
  opacity_bias: -2.0
  scale_bias: 0.02
  scale_scale: 0.003
  xyz_scale: 0.1
  xyz_bias: 0.0
  depth_scale: 1.0
  depth_bias: 0.0
  network_without_offset: false
  network_with_offset: true
  attention_resolutions:
  - 16
  num_blocks: 4
  cross_view_attention: true
  base_dim: 128
  isotropic: false

logging:
  ckpt_iterations: 1000
  val_log: 10000
  loss_log: 10
  loop_log: 10000
  render_log: 10000
```

Listing 21: Default YAML config for training

All training in Splatter Image is performed with the PyTorch Adam optimizer. The betas and learning rate and EMA toggle hyperparameters are initialized based on the training configuration, which as seen in the default config listing above, have default values of:

Betas: (0.9, 0.999)

Learning Rate: 0.00005

EMA: On

Exponential Moving Average (EMA) stabilizes learning and improves generalization by smoothing out weight changes during backpropagation.

The three pretrained prior models available on HuggingFace were trained starting from the original pretrained Splatter Image models following a graft. Those models were then trained for 60,000 additional iterations, with the LPIPS metric disabled.

In order to isolate the parameter-efficient adaptation due to LoRA, we used a dedicated training script (`train_network_lora.py`). While preserving the data pipeline and training loop architecture as the standard framework, it is necessary to enforce strict freezing where the gradients for the U-Net backbone are disabled. Optimisation is therefore restricted solely to the injected low-rank matrices (as described in **2.2.1 Low-Rank Adaptation (LoRA)**). All LoRA models were fine-tuned using the same optimizer settings as the baseline, but with LPIPS enabled. Training was conducted for 10,000 iterations, as discussed later, with checkpoints saved every 1000 iterations to handle restarts.

2.6 Testing and Validation Procedures

As discussed in previous sections, our changes often include a variety of runtime assertions to verify correctness, an example can be found in our input preparation changes, where assertions are performed to check if every prior requested for training is available in the training/test/validation datasets requested.

```
if "use_pred_depth" in model_cfg.data and model_cfg.data.use_pred_depth:
    assert model_cfg.data.category == "cars_priors", "Dataset does not have predicted maps!"
    input_images = torch.cat([input_images,
                    data["pred_depths"][:, :model_cfg.data.input_images, ...]],
                    dim=2)
if "use_pred_normal" in model_cfg.data and model_cfg.data.use_pred_normal:
    assert model_cfg.data.category == "cars_priors", "Dataset does not have predicted maps!"
    input_images = torch.cat([input_images,
                    data["pred_normals"][:, :model_cfg.data.input_images, ...]],
                    dim=2)
```

Listing 22: Excerpt from Splatter Image input tensor preparation code

Notebooks performing unit and integration tests were additionally written. Namely, 3 such notebooks can be found in the `testing` folder of the `3DGS-priors` repository.

The `eval_test.ipynb` notebook performs tests to verify if the modified evaluation script successfully performs evaluation both on previous datasets (such as ShapeNet cars) and our new dataset `cars_priors`. It does so by running correctly configured evaluations and checking whether the resulting `scortes.txt` files exist.

The `graft_test.ipynb` notebook performs tests checking whether the `graft_weights_with_channel_expansion` function returns successfully, for a variety of model configurations. It additionally verifies whether an automatically grafted model's state dictionary exactly matches the state dictionary of a manually grafted reference model.

The `dataloader_test.ipynb` notebook tests our custom PyTorch Dataset and ready generated `cars_priors` data. It does so by loading both the previous reference `srn.py` Dataset and our new `srn_priors.py` Dataset, walking both in an ordered mannered using Torch DataLoaders with shuffling disabled, and then comparing the resulting batches, which in our case means comparing common dictionary entries, where common key/-values entries should be identical.

All tests in these notebooks were run on Google Colab Pro+, using an A100 backend, and passed successfully; each notebook should have visible cell outputs showing this. No additional testing libraries or frameworks were used.

Chapter 3: Experiments and Evaluation

3.1 Datasets

[Explain the datasets utilized: what they contain, why they are utilized, assumptions, limitations, possible extensions.]
The standard benchmark for evaluating single-view 3D reconstruction is ShapeNet-SRN [43], hence we used this to test and evaluate our main model implementation. For this dataset, we specifically use the "Car" class, which used the "car" class of ShapeNet v2 [25] with 2.5k 3D CAD model instances. The SRN dataset was generated by disabling transparencies and specularities and training on 50 observations of each instance at a resolution of 128×128 pixels, with camera poses being randomly generated on a sphere with the object at the origin. A limitation of this dataset is the lack of subject variety in the dataset as the model may end up overfitting to cars. A possible extension to address this limitation could be to include other classes in the ShapeNet-SRN database to make sure that the model can still generalise to other types of objects.
We also use an extension of this dataset at `https://github.com/Xharlie/ShapenetRender_more_variation` which was produced by [24], which presents a Deep Implicit Surface Network to generate a 3D mesh from a 2D image by predicting the underlying signed distance fields. [24] generated a 2D dataset composed of renderings of the models in ShapeNet Core [25]. For each mesh model, the dataset provides 36 renderings with smaller variation and 36 views with larger variation (bigger yaw angle range and larger distance variation). The object is allowed to move away from the origin, which provides more degrees of freedom in terms of camera parameters, and the "roll" angle of the camera is ignored since it was deemed very rare in real-world scenarios. The images were rendered at a higher resolution of 224×224 pixels and were paired with a depth image, a normal map and an albedo image as shown in figure 11. This dataset was mainly used as a ground truth to evaluate the generation of geometry priors (e.g. normal map and depth map). The ground truth subset also included an alpha channel, which allowed us to determine a deterministic binary marks. This allowed us to quantitatively evaluate the performance of our Salient Object Detection (SOD) models. The samples within this dataset are categorised into 'Easy' and 'Hard', which allowed us to evaluate the performance of geometry prior generation on different levels of difficulty.
A limitation of this dataset would be its small size since only 72 samples are available for us to use, such that the performance of geometry prior generation may not be evaluated correctly. However, in the same GitHub repository, the script to generate these images from the ShapeNet Core dataset is provided, so a possible extension given more time could be to include more images by running the script on other objects in the ShapeNet Core dataset. Another limitation is that some of the depth maps provided were flawed as shown in figure 17, so using them as ground truths to evaluate the performance of the depth map generator model was not ideal. In the future, evaluation could benefit from regenerating these depth maps to be more accurate, perhaps by exploring a different method of depth map generation from the 3D model provided in the ShapeNet Core dataset.

3.2 Training and testing results

[Explain the training and testing results with graphs and elaborating on why they make sense, what could be improved.]

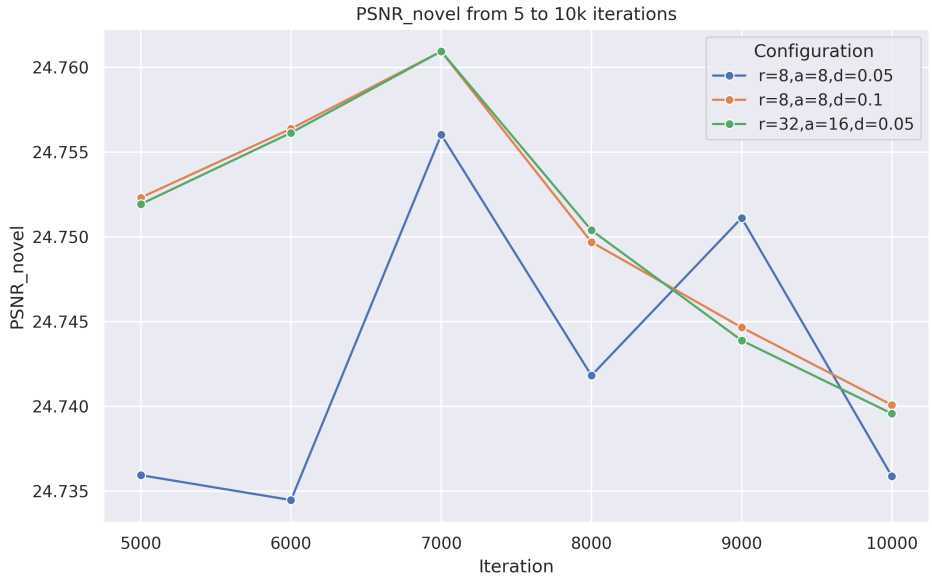


Figure 26: Validation PSNR from 5000-10000 iterations

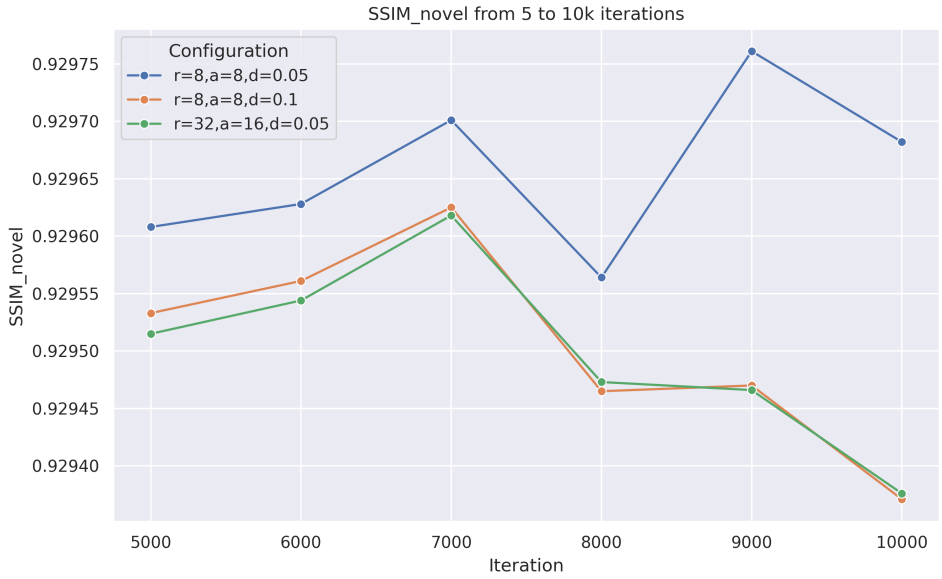


Figure 27: Validation SSIM from 5000-10000 iterations

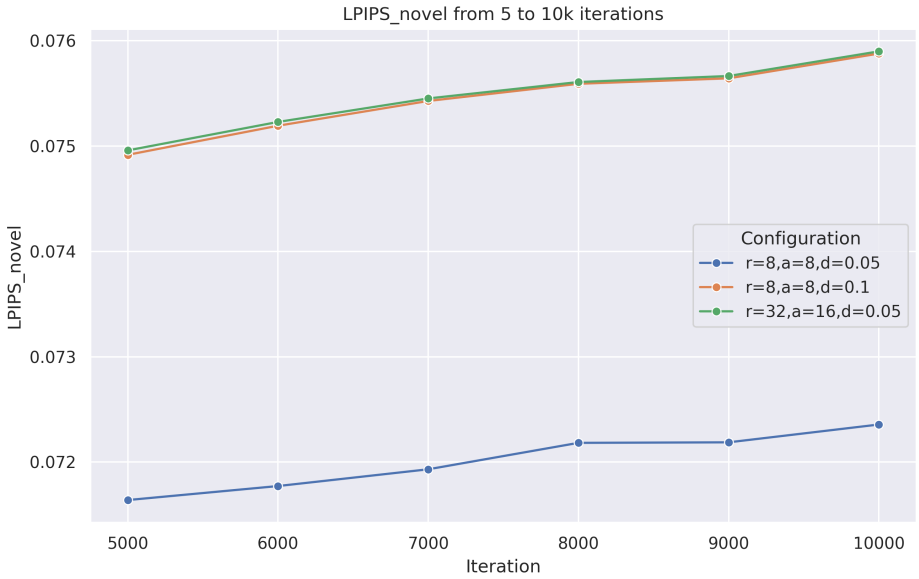


Figure 28: Validation LPIPS from 5000-10000 iterations

The metrics introduced here are discussed further in **Section 3.4.3: LoRA Experiment Results**.
When considering LoRA, the validation metrics from 5 to 10,000 iterations of finetuning reveal a distinct plateau, as we can note the y-axis scale as being extremely tight for the metric scores. Despite testing different rank configurations, we reached what appears to be the convergence limit of the weights early in the finetuning process. This behavior could be attributed to the specific initialisation used, where the adapters are initialised with B as zero, allowing a quick plateau (in a few thousand iterations) if the base model's weights were already residing in or close to a local optimum, and this the loss gradients with respect to the LoRA parameters are negligible. Similarly, the similar trajectories between the low and high rank configurations suggest that model capacity may not be the bottleneck.
To verify if this plateau is a hard convergence limit, we could explore several improvements, such as using a larger learning rate for LoRA, or restricting LoRA adaptation to other layers in the U-Net (this could mean considering adding LoRA to the cross-attention blocks in the U-Net). Performing a full finetuning experiment would also help indicate whether the bottleneck comes from the low-rank enforcement or the data used.

3.3 Qualitative results

[Show in figures and explain visual results. Include different interesting cases covering different aspects/ limitations/ dataset diversity. If not converged, explain what we can expect once converged. Include any other didactic examples here.]

To qualitatively assess the reconstruction quality, we can evaluate our models on two categories of viewpoints as defined in the evaluation protocol: conditional Views (the input angle) and novel views (unseen angles).
Figure (Figure 29) looks at conditional views which correspond to the exact camera pose used as input to the model. We expect high performance here, as the model should successfully encode input image features into the Gaussian representation for the object.
As observed below, both tests visually achieve very similar results in comparison to the ground truth. We see an accurate silhouette, with sharp features.

(a) Geometry Priors Model (Input View)

(b) LoRA Model (Input View)

Figure 29: Conditional View Comparison of the input view reconstruction. Left: Ground Truth, Right: Prediction.

The primary challenge of single-view reconstruction lies in novel view synthesis (which is consequently the focus of **Section 3.4: Quantitative Results**), where the model must infer occluded geometry without direct data. Figure 30 illustrates this capability.
The results in the following for a LoRA-adapted model is for a configuration of rank 32, α 16, and dropout 0.05.

(a) Geometry Priors Model: Novel View (Left: GT, Right: Pred)

(b) LoRA Model: Novel View (Left: GT, Right: Pred)

Figure 30: Novel View Comparison considering unseen angles.

3.4 [Optional] Quantitative Results

[A table and associated explanations for quantitative results.]

3.4.1 Metrics

For novel view synthesis evaluation, PSNR (Peak Signal-to-Noise Ratio), SSIM (Structural Similarity Index) and LPIPS (Learned Perceptual Image Patch Similarity) are standard metrics used to evaluate image quality from different perspectives[9]. PSNR is calculated using the mean squared error and is commonly used to quantify reconstruction quality, such that a high PSNR suggests higher reconstruction quality. Meanwhile, SSIM depends on 3 metrics mimicking the human visual perception system: luminance, contrast and structure, where an SSIM score close to 1 indicates high similarity while a score closer to -1 indicates low similarity. Lastly, unlike pixel-wise metrics like PSNR and SSIM that assume pixel independence, LPIPS measures the perceptual similarity between images by comparing their features extracted from a deep neural network, where a low LPIPS score indicates that the compared images are perceptually similar to humans.

To evaluate whether integrating different geometry priors into the model is effective, we performed a set

of experiments testing different combinations of geometry priors. We measured the performance using PSNR, SSIM and LPIPS.

3.4.2 Ablation Study Results

We train our models with different combinations of added geometry priors: one with depth maps, one with normal maps and one with both. We compare these models with our baseline.

It is important to note that the original Splatter Image implementation recommends that single-view models be trained in two stages, first without LPIPS, followed by fine-tuning with LPIPS. However, due to the large computational overhead caused by fine-tuning with LPIPS and our lack of computational resources and time, we did not run the second stage with LPIPS. Hence, although we include LPIPS in our metrics in this section, we do not use it to evaluate model performance as it is not an accurate assessment.

Table 4: Significance testing for metrics of augmented models vs baseline (to 3 s.f.)

Added Geometry Priors	**Metric**	**P-value**
Depth Only	PSNR	0.000247
	SSIM	0.000273
	LPIPS	0.0311
Normals Only	PSNR	0.000247
	SSIM	0.000187
	LPIPS	0.0369
Both	PSNR	0.000100
	SSIM	0.000100
	LPIPS	0.0385

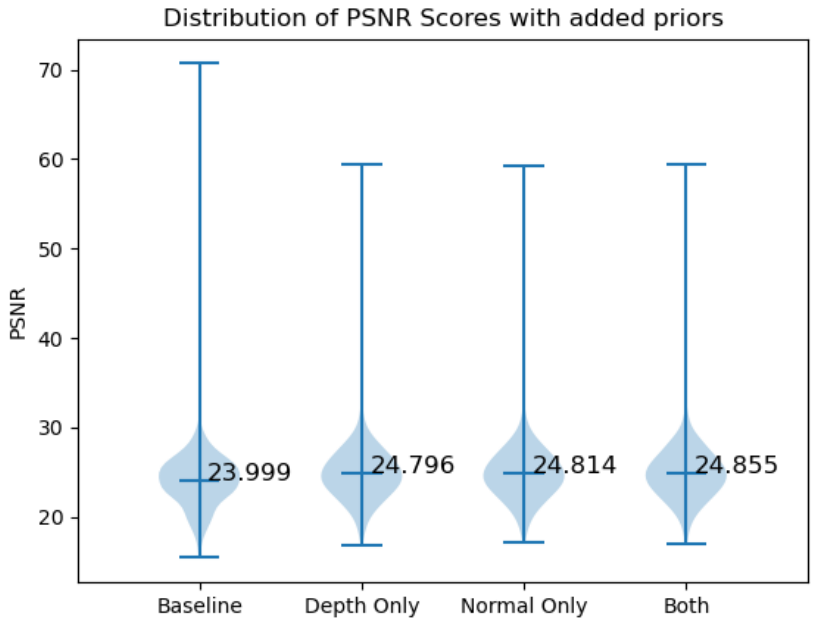


Figure 31: PSNR distribution for different combinations of added geometry priors

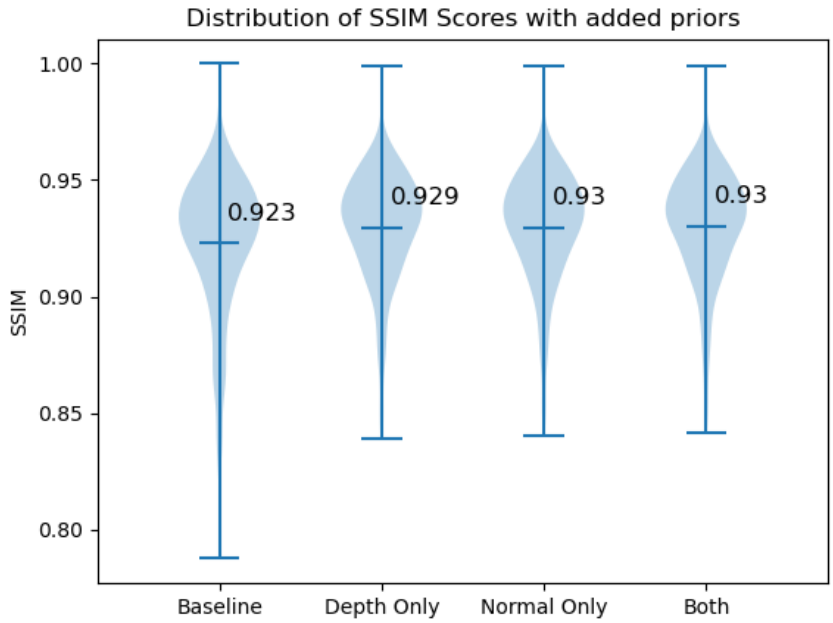


Figure 32: SSIM distribution for different combinations of added geometry priors

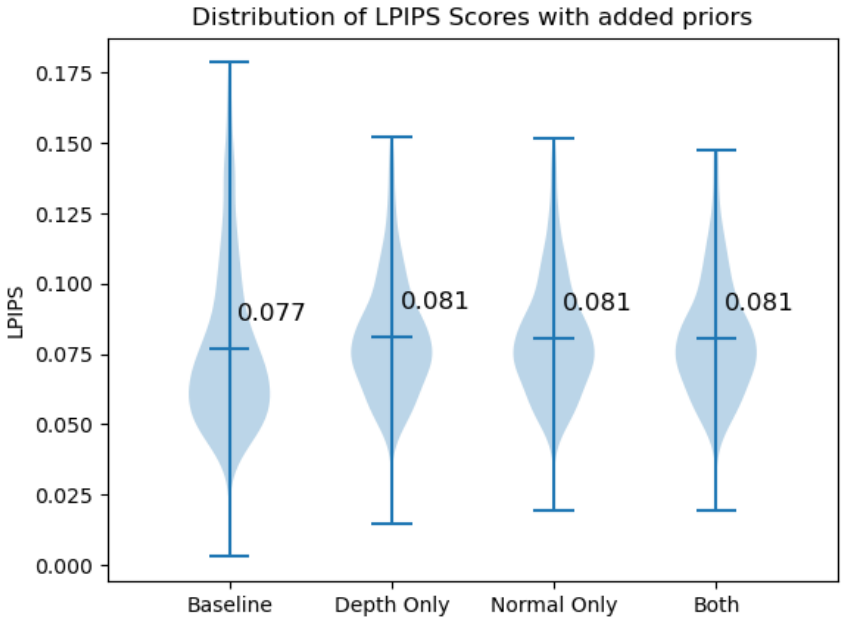


Figure 33: LPIPS distribution for different combinations of added geometry priors

Table 5: Results for ablation study

Added priors	**PSNR ↑**	**SSIM ↑**	**LPIPS (not considered) ↓**
Baseline	23.999	0.923	0.077
Depth Only	24.796	0.929	0.081
Normal Only	24.814	**0.93**	0.081
Both	**24.855**	**0.93**	0.081

We conduct a two-sided significance test with non-parametric resampling to calculate the P-value, which is the probability that the metrics of the augmented model are not significantly different from that of the original model and any difference is only due to random variation. We select the power of the test to be 0.05. Since all P-values are less than 0.05 from table 4, we conclude that the metrics of augmented model significantly differ from that of the original model.
Adding combinations of depth and normal priors to augment the model with geometry priors seems to make the model perform more consistently, as shown by the lower variance in all metrics as compared to the baseline in figures 31, 32 and 33. All augmented models have a higher mean PSNR and SSIM from figures 31 and 32 respectively which suggests better image reconstruction quality, with the model adding both depth and normal priors performing the best. As expected, all augmented models are worse than the baseline with regards to LPIPS (as shown in figure 33), since these models were not trained with LPIPS.

3.4.3 LoRA Experiment Results

To evaluate whether Low-Rank Adaptation (as described in **Section 2.3.6: Manual LoRA Integration**) is effective, we performed a set of fine-tuning experiments on the ShapeNet-SRN Cars dataset. As mentioned in **Section 2.2.2: Low-Rank Adaptation (LoRA)**, LoRA behaviour is primarily controlled by three hyperparameters: rank (r), alpha (α), and dropout (d). We tested multiple configurations by varying these hyperparameters while keeping all other components constant, measuring performance using PSNR, SSIM, and LPIPS.
Table 6 presents the quantitative results for novel view synthesis early in the process (after 5,000 iterations).

Table 6: Results for LoRA at 5,000 iterations. All configurations cluster tightly around 24.75 for PSNR.

Configuration	**PSNR ↑**	**SSIM ↑**	**LPIPS ↓**
$r = 8, \alpha = 8, d = 0.05$	24.74	0.930	0.072
$r = 32, \alpha = 32, d = 0.5$	24.74	0.929	0.075
$r = 32, \alpha = 16, d = 0.5$	24.75	0.930	0.075
$r = 16, \alpha = 16, d = 0.0$	24.77	0.930	0.076

We continued training for 10,000 iterations. Table 7 details the results for three distinct model configurations at the end of this longer run. The differences between configurations become slightly more pronounced after 10k iterations compared to the 5k mark.

Table 7: Comparison of different LoRA models after 10,000 iterations.

Configuration	**PSNR ↑**	**SSIM ↑**	**LPIPS ↓**
$r = 8, \alpha = 8, d = 0.05$	24.81	**0.931**	**0.068**
$r = 32, \alpha = 16, d = 0.05$	24.83	**0.931**	0.072
$r = 8, \alpha = 8, d = 0.1$	**24.84**	**0.931**	0.072

Here we can see distribution plots for the scores files:

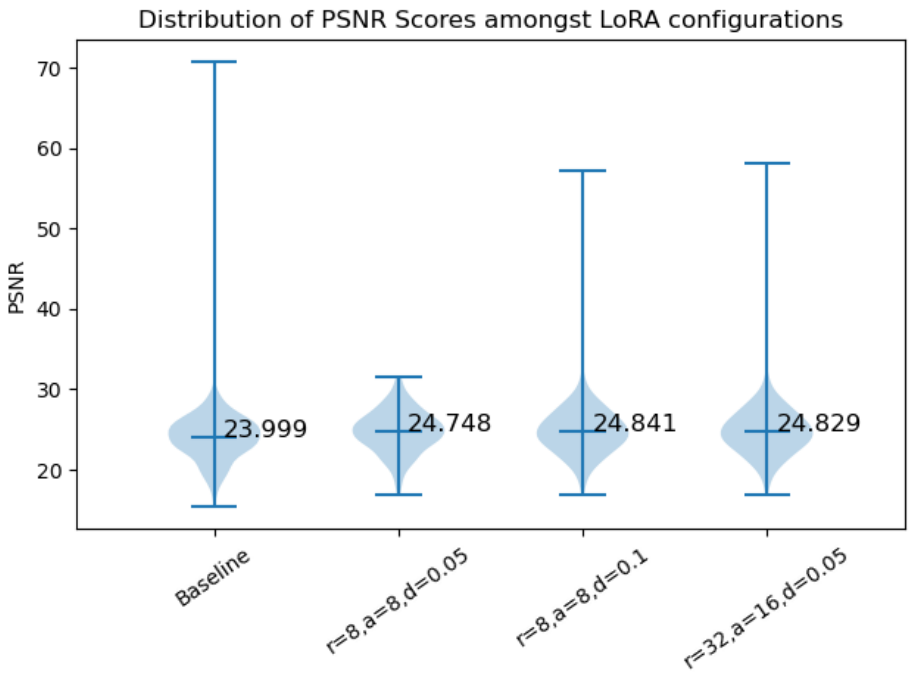


Figure 34: PSNR distribution for different LoRA configurations

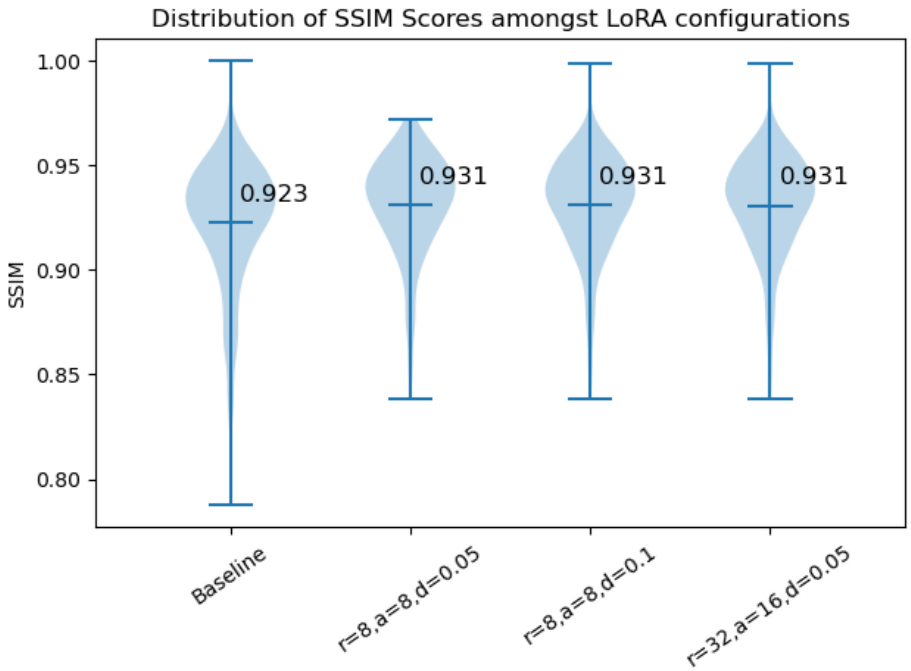


Figure 35: SSIM distribution for different LoRA configurations

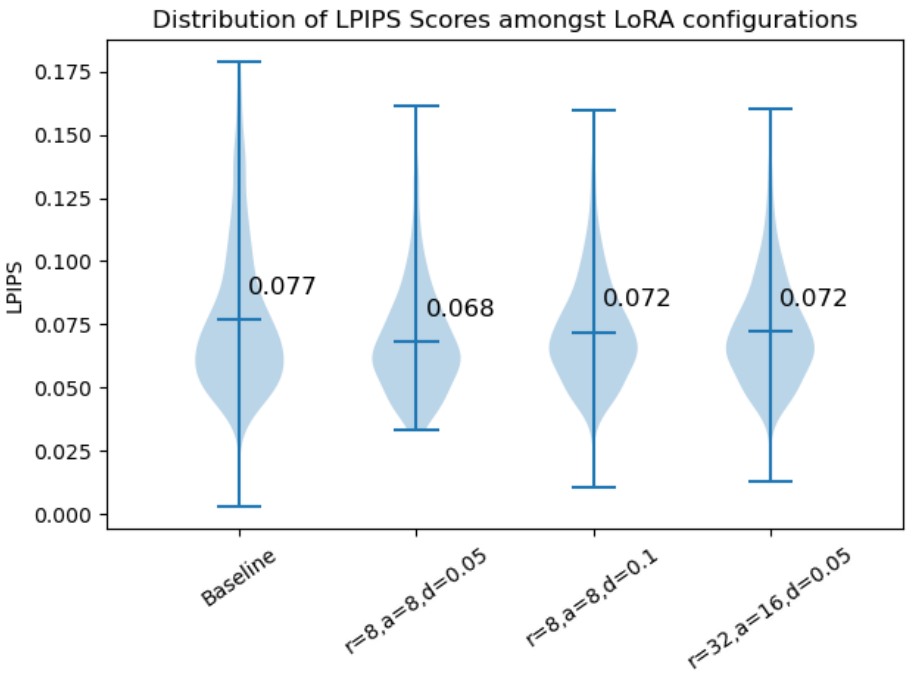


Figure 36: LPIPS distribution for different LoRA configurations

Table 8: Significance testing for metrics of different LoRA configurations vs baseline (to 3 s.f.)

Configuration	**Metric**	**P-value**
$r = 8, \alpha = 8, d = 0.05$	PSNR	0.00141
	SSIM	0.0000600
	LPIPS	0.0000133
$r = 32, \alpha = 16, d = 0.05$	PSNR	0.000180
	SSIM	0.000
	LPIPS	0.00408
$r = 8, \alpha = 8, d = 0.1$	PSNR	0.000107
	SSIM	0.00000670
	LPIPS	0.00309

We conduct a two-sided significance test with non-parametric resampling to calculate the P-value, similar to **Section 3.4.2: Ablation Study Results**. We select the power of the test to be 0.05. Since all P-values are less than 0.05 from table 8, we conclude that the metrics for models with LoRA significantly differ from that of the original model.

Chapter 4: Conclusions and Future Directions

4.1 Conclusions

[Summarize what the project was about and the main conclusions.]
Altogether, we managed to successfully augment the baseline Splatter Image implementation with our geometry priors being normal and depth maps. We fed these priors to the model both through the input channels and by using LoRA, and found that both approaches resulted in a higher reconstruction quality than the

baseline as measured by metrics PSNR, SSIM and LPIPS. By making these lightweight improvements while preserving Splatter Image's key architecture, we increase its quality while still allowing it to maintain its original computational efficiency with its 3D Gaussian Splatting approach.
Through our ablation study, we also saw that Splatter Image performed best when augmented with both depth and normal maps, showing that both priors are very useful for informing the model about how to reconstruct unseen parts of the input image.

4.2 Discussion of limitations

[Explain the limitations of your technique. You may want to refer to previous sections or show figures on the limitations.]

Our biggest limitations stemmed from a lack of time, compute resources, and tooling problems. Splatter Image, despite its relative high performance for a deep learning reconstruction model, still requires powerful hardware to run. Not only does it need access to powerful hardware, but it needs access for lengthy periods of time, as to perform tasks like training. In terms of tooling and hardware, we used Google Colab Pro+ to give us access to A100 GPUs and 24 hour execution notebooks; despite these promises, we found that our notebooks running overnight were very often randomly killed by Google, this caused the complete loss of the VM/notebook and all associated progress (namely model training progress). This meant we could not reliably train models to achieve comparable results to the original pretrained models, or truly show the effectiveness of our modifications, where for comparison, the Splatter Image provided pretrained models were trained continuously on A6000 GPUs over a period of 7 days. Should we have reached adequate training time, the quality of our selected prior models would have likely been the limiting factor for improvements to Splatter Image, as Splatter Image can only be as good as the quality of the input (principle of garbage-in-garbage-out).

Lack of time and compute also meant we could not afford to generate and test additional priors other than depths and normals maps, despite having ready implementations of prior generation scripts like the `generate_segmentation.py` script.

Another set of limitations came in the form of poor documentation. During research of prior models we often struggled to run the models in the first place, due to incorrectly provided package requirements, serious (or in some cases total) lack of instructions on how to launch and use the models, or no reference implementations being publicly provided at all, as was often the case with research papers. This made our search for priors models difficult and time consuming, on a already very time stretched project.

We were also given access to the reference SRN cars dataset at a very late stage in the project, which did not leave us enough time to implement desired things like calculating our own reliable ground truths, or being able to implement additional reconstruction metrics like Chamfer distance, where we could have generated point clouds from SRN car models necessary for the calculation.

4.3 Future Directions

[State a few future directions for research and development. These typically follow from the discussion on limitations.]

One avenue for future work is generating and testing more geometry priors for future research. Due to the lack of time, there were some priors that we deemed not worth investigating. For example, in **Section 2.3.1: Planes and Normal Maps Exploration**, we discuss how generating predicted scene planes for input images and using them as added priors to augment the model seemed unlikely to help guide reconstruction. In future studies, we could challenge this assumption by generating this prior and testing it alongside other geometry priors in the ablation study. We could also use the SOD models we explored in **Section 2.3.3: Segmentation and Salient Object Detection Exploration** to generate binary masks and use them as priors to augment our model and see how they affect model performance. In general, since Splatter Image

has most trouble generating sections of novel views that are obscured from the original input image, we could focus on generating and testing more priors oriented to hidden areas.
Some of these priors may be of a non-image format, making them incompatible with simply being stacked into the model input channels. Such multimodal inputs such as classifications from segmentation models require a more permanent or multi-embedding capable multimodal input method, over the current basic single FiLM input integration. Better FiLM support or mechanism like cross-attention could be implemented in the future, and then these approaches could be comapred to see which is most useful for Splatter Image.
We could also investigate the integration of explicit geometric loss functions to supplement the current standard losses. A specialised 'rendering loss' could be introduced that targets depth discontinuities, especially following on from the discussion in **Section 2.3.2: Depth Map Exploration**. The model could be penalised for failing to reconstruct structural boundaries (by applying edge operators on rendered views and comparing them with ground truth edge maps if possible). This strategy was successfully demonstrated in [44]. This loss can also be dynamically weighted based on the size of the 3D Gaussian's covariance matrices, as 'smaller' Gaussians, with smaller covariance determinants, generally represent fine details. If this loss is prioritised, i.e. we prioritise areas contributed to by small scale Gaussians, the optimisation process could be forced to resolve fine details, rather than being treated as noise or being approximated by larger Gaussians.
Another avenue for future work could be to train and evaluate the models with a more diverse set of datasets. As mentioned in **Section 3.1: Datasets**, our training and tested was conducted with the ShapeNet-SRN cars dataset, which may cause our models to overfit to cars and not generalise well to other types of objects. We could try to address this by training the model with other datasets with different objects, such as the Objaverse [45][46] datasets and the CO3D [23] dataset. With more computational resources, we could also explore increasing training times for the models to see if their performances can be improved further with more iterations of training.
Another possible extension could be to try using alternative metrics for 3D reconstruction such as Chamfer distance, where we could generate point clouds from SRN models and compare them to the 3D Gaussians predicted by Splatter Image.

4.4 Project Contributions

Report Writing Contributions:
Section 1.1: Kacper and Alex
Section 1.2: Kacper and Alex
Section 1.3: Kacper
Section 2.1: Alex
Section 2.2: Kacper and Radhika
Section 2.3: Kacper and Radhika and Alex
Section 2.4: Kacper and Alex
Section 2.5: Kacper and Radhika
Section 2.6: Kacper
Section 3.1: Alex and Radhika
Section 3.2: Radhika and Alex
Section 3.3: Radhika
Section 3.4: Radhika and Alex
Section 4.1: Alex
Section 4.2: Kacper and Alex
Section 4.3: Kacper and Radhika and Alex
Image and citation collection: Kacper and Radhika and Alex

Presentation Contributions:
Slides and Recording: Kacper and Radhika and Alex
Video Editing: Alex

Technical Contributions:
Depths exploration: Radhika

Edge operators exploration: Radhika
Segmentation exploration: Radhika
Normals exploration: Alex
Planes exploration: Alex
Splatter Image setup and bugfixes: Radhika and Alex
Splatter Image architectural modification (Grafting, Channel changes, FiLM): Kacper
Splatter Image LoRA integration: Radhika
Splatter Image Training Modification: Kacper and Radhika
Splatter Image Eval Modification: Kacper
Optimised depth generation: Kacper
Optimised normals generation: Kacper and Alex
Optimised segmentation generation: Radhika
Splatter Image `cars_priors` custom Dataset: Kacper
HuggingFace dataset data pipeline: Kacper
Results processing code: Alex
Graphing code: Alex
Testing custom Dataset: Kacper
Testing evaluation: Kacper and Radhika
Testing prior configurations: Kacper
Testing LoRA configurations: Radhika

References

[1] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020.

[2] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering, 2023.

[3] Shengji Tang, Weicai Ye, Peng Ye, Weihao Lin, Yang Zhou, Tao Chen, and Wanli Ouyang. Hisplat: Hierarchical 3d gaussian splatting for generalizable sparse-view reconstruction, 2024.

[4] Stanislaw Szymanowicz, Christian Rupprecht, and Andrea Vedaldi. Splatter image: Ultra-fast single-view 3d reconstruction, 2024.

[5] Jianghao Shen, Nan Xue, and Tianfu Wu. A pixel is worth more than one 3d gaussians in single-view 3d reconstruction, 2024.

[6] Zhizhong Kang, Juntao Yang, Zhou Yang, and Sai Cheng. A review of techniques for 3d reconstruction of indoor environments. *ISPRS International Journal of Geo-Information*, 9(5), 2020.

[7] Andrea Romanoni, Amaël Delaunoy, Marc Pollefeys, and Matteo Matteucci. Automatic 3d reconstruction of manifold meshes via delaunay triangulation and mesh sweeping, 2016.

[8] Timothy Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30:854–879, 10 2006.

[9] Jiahui Zhang, Yuelei Li, Anpei Chen, Muyu Xu, Kunhao Liu, Jianyuan Wang, Xiao-Xiao Long, Hanxue Liang, Zexiang Xu, Hao Su, Christian Theobalt, Christian Rupprecht, Andrea Vedaldi, Kaichen Zhou, Paul Pu Liang, Shijian Lu, and Fangneng Zhan. Advances in feed-forward 3d reconstruction and view synthesis: A survey, 2025.

[10] Tianyi Gong, Boyan Li, Yifei Zhong, and Fangxin Wang. Exscene: Free-view 3d scene reconstruction with gaussian splatting from a single image, 2025.

[11] Hanwen Liang, Junli Cao, Vidit Goel, Guocheng Qian, Sergei Korolev, Demetri Terzopoulos, Konstantinos N. Plataniotis, Sergey Tulyakov, and Jian Ren. Wonderland: Navigating 3d scenes from a single image, 2025.

[12] Yuxin Wang, Qianyi Wu, and Dan Xu. F3d-gaus: Feed-forward 3d-aware generation on imagenet with cycle-aggregative gaussian splatting, 2025.

[13] Junlin Hao, Peiheng Wang, Haoyang Wang, Xinggong Zhang, and Zongming Guo. Gaussvideodreamer: 3d scene generation with video diffusion and inconsistency-aware gaussian splatting, 2025.

[14] Zi-Xin Zou, Zhipeng Yu, Yuan-Chen Guo, Yangguang Li, Ding Liang, Yan-Pei Cao, and Song-Hai Zhang. Triplane meets gaussian splatting: Fast and generalizable single-view 3d reconstruction with transformers, 2023.

[15] Shangchen Zhou, Chongyi Li, Kelvin C. K. Chan, and Chen Change Loy. Propainter: Improving propagation and transformer for video inpainting, 2023.

[16] Yuwei Guo, Ceyuan Yang, Anyi Rao, Zhengyang Liang, Yaohui Wang, Yu Qiao, Maneesh Agrawala, Dahua Lin, and Bo Dai. Animatediff: Animate your personalized text-to-image diffusion models without specific tuning, 2024.

[17] Eric R. Chan, Connor Z. Lin, Matthew A. Chan, Koki Nagano, Boxiao Pan, Shalini De Mello, Orazio Gallo, Leonidas Guibas, Jonathan Tremblay, Sameh Khamis, Tero Karras, and Gordon Wetzstein. Efficient geometry-aware 3d generative adversarial networks, 2022.

[18] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[19] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models, 2022.

[20] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.

[21] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *CoRR*, abs/2106.09685, 2021.

[22] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[23] Jeremy Reizenstein, Roman Shapovalov, Philipp Henzler, Luca Sbordone, Patrick Labatut, and David Novotný. Common objects in 3d: Large-scale learning and evaluation of real-life 3d category reconstruction. *CoRR*, abs/2109.00512, 2021.

[24] Qiangeng Xu, Weiyue Wang, Duygu Ceylan, Radomir Mech, and Ulrich Neumann. Disn: Deep implicit surface network for high-quality single-view 3d reconstruction. In *NeurIPS*, 2019.

[25] Angel X. Chang, Thomas A. Funkhouser, Leonidas J. Guibas, Pat Hanrahan, Qi-Xing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. Shapenet: An information-rich 3d model repository. *CoRR*, abs/1512.03012, 2015.

[26] Gwangbin Bae, Ignas Budvytis, and Roberto Cipolla. Estimating and exploiting the aleatoric uncertainty in surface normal estimation. In *International Conference on Computer Vision (ICCV)*, 2021.

[27] Angela Dai, Angel X. Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 2017.

[28] Jingwei Huang, Yichao Zhou, Thomas Funkhouser, and Leonidas Guibas. Framenet: Learning local canonical frames of 3d surfaces from a single rgb image. *arXiv preprint arXiv:1903.12305*, 2019.

[29] Pushmeet Kohli Nathan Silberman, Derek Hoiem and Rob Fergus. Indoor segmentation and support inference from rgbd images. In *ECCV*, 2012.

[30] Xiaojuan Qi, Renjie Liao, Zhengzhe Liu, Raquel Urtasun, and Jiaya Jia. Geonet: Geometric neural network for joint depth and surface normal estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 283–291, 2018.

[31] Xiaojuan Qi, Zhengzhe Liu, Renjie Liao, Philip HS Torr, Raquel Urtasun, and Jiaya Jia. Geonet++: Iterative geometric neural network with edge-aware refinement for joint depth and surface normal estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[32] Saeed Mahmoudpour and Manbae Kim. Chapter 10 - a study on the relationship between depth map quality and stereoscopic image quality using upsampled depth maps. In Leonidas Deligiannidis and Hamid R. Arabnia, editors, *Emerging Trends in Image Processing, Computer Vision and Pattern Recognition*, pages 149–160. Morgan Kaufmann, Boston, 2015.

[33] Xinwei Liu, Marius Pedersen, and Renfang Wang. Survey of natural image enhancement techniques: Classification, evaluation, challenges, and perspectives. *Digital Signal Processing*, 127:103547, 2022.

[34] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. `https://github.com/facebookresearch/detectron2`, 2019.

[35] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything. *arXiv:2304.02643*, 2023.

[36] Daniel Gatis. rembg. `https://github.com/danielgatis/rembg`, 2025. Version 2.0.66.

[37] Taehun Kim, Kunhee Kim, Joonyeong Lee, Dongmin Cha, Jiho Lee, and Daijin Kim. Revisiting image pyramid structure for high resolution salient object detection. In *Proceedings of the Asian Conference on Computer Vision*, pages 108–124, 2022.

[38] Peng Zheng, Dehong Gao, Deng-Ping Fan, Li Liu, Jorma Laaksonen, Wanli Ouyang, and Nicu Sebe. Bilateral reference for high-resolution dichotomous image segmentation. *CAAI Artificial Intelligence Research*, 3:9150038, 2024.

[39] Radhakrishna Achanta, Sheila Hemami, Francisco Estrada, and Sabine Susstrunk. Frequency-tuned salient region detection. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1597–1604, 2009.

[40] Ming-Ming Cheng, Niloy J. Mitra, Xiaolei Huang, Philip H. S. Torr, and Shi-Min Hu. Global contrast based salient region detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(3):569–582, 2015.

[41] Ali Borji, Ming-Ming Cheng, Huaizu Jiang, and Jia Li. Salient object detection: A benchmark. *IEEE Transactions on Image Processing*, 24(12):5706–5722, December 2015.

[42] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, Benjamin Bossan, and Marian Tietz. PEFT: State-of-the-art parameter-efficient fine-tuning methods. `https://github.com/huggingface/peft`, 2022.

[43] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations, 2020.

[44] Yuanhao Gong. Eggs: Edge guided gaussian splatting for radiance fields, 2024.

[45] Matt Deitke, Dustin Schwenk, Jordi Salvador, Luca Weihs, Oscar Michel, Eli VanderBilt, Ludwig Schmidt, Kiana Ehsani, Aniruddha Kembhavi, and Ali Farhadi. Objaverse: A universe of annotated 3d objects, 2022.

[46] Matt Deitke, Ruoshi Liu, Matthew Wallingford, Huong Ngo, Oscar Michel, Aditya Kusupati, Alan Fan, Christian Laforte, Vikram Voleti, Samir Yitzhak Gadre, Eli VanderBilt, Aniruddha Kembhavi, Carl Vondrick, Georgia Gkioxari, Kiana Ehsani, Ludwig Schmidt, and Ali Farhadi. Objaverse-xl: A universe of 10m+ 3d objects, 2023.