EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Faculty of Science

Autonomous Vision Group

Researchproject

Novel View Synthesis on KITTI-360

Eberhard Karls University of Tübingen
Faculty of Science
Wilhelm-Schickard Institute
Autonomous Vision Group
Bernhard Jaeger, bernhard.jaeger@student.uni-tuebingen.de, 2021

Period of processing: 01.10.2020-31.01.2021

Supervisor: Prof. Dr. Andreas Geiger, University of Tübingen
Second supervisor: Dr. Yiyi Liao, University of Tübingen

Declaration of Authorship

Hereby, I declare that I have composed the presented report independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such. This work was not used in any other study program.

Bernhard Jaeger (Matrikelnummer 5483684), January 31, 2021

Abstract

Groups working in the field of Novel View Synthesis evaluate their methods on their own respective scenes. This makes comparing methods and evaluating the progress of the community hard. We propose to address this issue using the KITTI-360 dataset as a benchmark. To that end we investigate two baselines on the dataset Free View Synthesis and Point Cloud Rendering. We also discuss a potential new field Novel Label Synthesis.

Acknowledgments

I thank Andreas Geiger and Yiyi Liao for supervision on this project, Shrisha Bharadwaj and Gernot Riegler for helpful discussions and the Stackoverflow community for their help on technical problems.

Contents

1 Introduction

1.1 Problem Statement

Novel View Synthesis is the task of generating new images of a scene given a set of input images from that scene. Most method described as Novel View Synthesis methods use machine learning techniques, specifically deep neural networks to accomplish their task. These methods have emerged over the last couple of years. Perhaps due to the recent emergence of the field there is no common task framework [Don17] available for this task yet. Groups working in the field instead evaluate their methods on their individual data sets. These evaluations can vary significantly in different ways such as size of the scene investigated, maximum camera movement or number of input images. This makes it hard to compare different methods and evaluate the progress of the scene as a whole. As an example of this problem see [RK20] where the methods Neural Radiance Fields [MST+20] and Local Light Field Fusion [MSC+19] were evaluated on scenes from the Tanks and Temples data set and shown to struggle. Since these limitations were not discussed in the respective papers proposing these methods it is hard to know whether a method will work before you try it out and hard to compare them just based on their reports.

One common way to address these problems is the common task framework mentioned earlier. A potential candidate for such a framework could be the KITTI-360 dataset [XKSG16a] [kit]. It contains over 320k raw images from unconstrained outdoor scenes recorded while driving a car. In this work we investigate and develop two baseline algorithms on KITTI-360 to serve as baselines for a benchmark.

Another advantage of the KITTI-360 data set is that it provides ground truth semantic segmentation labels. Since it is possible to generate novel RGB camera views of a scene one can easily imagine that the same thing should also be possible for semantic segmentation labels corresponding to those generated RGB views. We call this task Novel Label Synthesis and will investigate it more section 3.4.

In the following we will first review relevant literature in chapter 2. Works directly relevant to Novel View Synthesis will be discussed in section 2.1. Since Novel View Synthesis is embedded in the bigger fields of Computer Graphics, 3D Reconstruction and Deep Learning we will also discuss some works in those areas in section 2.2. Chapters 3 and 4 will discuss the methods we use and the results that they produced. The report will be concluded with a discussing in chapter 5.

2 Literature Review

2.1 Related Work

Creating novels views based on raw input images is not exactly a new idea. Already in the 1990's foundational work in this area has been published in the closely related field of Image Based Rendering. Two important works from that time where The Lumigraph [GGSC96] and Light Field Rendering [LH96]. They share a common idea which is to view the scene as a 5D plenoptic function. This function takes as input the 3D position (x, y, z) and 2D viewing direction described by the angles (θ, ϕ) and return the flow of light/ radiance. The radiance along a ray remains constant in empty space (and only changes when hitting an object). Therefore one of the dimensions can be said to contain only little information. These works therefore model this 5D function as 4D Lumigraph / light field. Input images are then seen as samples of 2D slices through these models and used to estimate them. One may view these works as predecessors of the modern Neural Radiance Fields approach and it's many variants (for a list of them see [DYC21]). One interesting detail is that these early work had to spend a lot of effort on memory compression, camera pose estimation (requiring special laboratory setups) and one might argue simplifying the problem by going from 5D to 4D. Due to advances in computational resources and camera pose estimation modern works don't seem to have to bother too much with these issues anymore.

Now fast forward 20 years one of the earlier Novel View Synthesis methods was called View Synthesis by Appearance Flow [ZTS+16]. Here they generate novel views by predicting the appearance flows between the input image and the target image via a Convolutional Neural Network (CNN). Appearance flows are vectors from pixels in the input image that map their color to a pixel in the output image. This already worked for images of small objects and to some degree also outdoor scenes. However there was still a lot of room for improvement in terms of image quality.

A common axis along which novel view synthesis methods differ is the representation they use to model the underlying scene (explicitly or implicitly). Representations that are used are meshes, point clouds, voxels, implicit representation and multi plane images.

Multi plane images (MPI) divide the scene into multiple parallel planes/RGBA images at different depths. An illustration of this can be seen in figure 2.1[ZTF+18]. Stereo magnification [ZTF+18] is a method that uses a fully convolutional neural

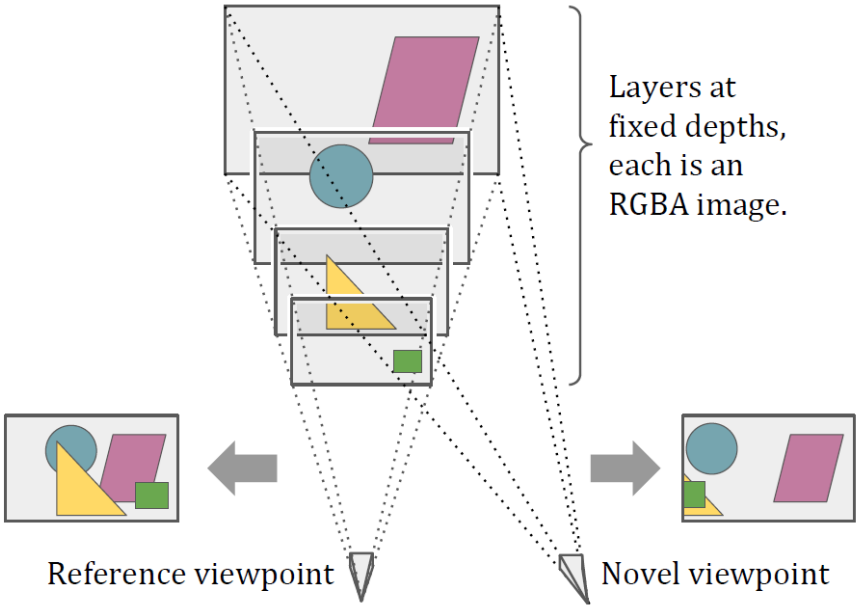

Figure 2.1: A multi plane image representation of a toy scene

network to predict the multi plane image representation. The synthesized image results shown in the paper are of very high quality. However the method was only evaluated on camera poses that were within a few centimeters of the original camera and the camera only moved along the plane of the input image. It is likely that if the camera moved depth wise and rotate the method would break down since the fronto-parallel assumption behind the multi plane image representation would be violated.

Deep View [FBD+19] is another method using the multi plane image representation. They increase the range the camera can move on the original image plane and their method does not require scene fine tuning. This is achieved using an iterative procedure where a Convolutional Neural Network first proposes an MPI and than iteratively improves it. Additionally the gradient decent used to optimize is also learned by a neural network.

Voxel based representations are the natural extension of pixels into the third dimension. A voxel represents a discretized volume of 3D space and is ordered on a regular grid. An advantage of this representation is that convolutional layers can be used on this representation directly. The downside is that due to the qubic memory requirements (curse of dimensionality) only a very coarse resolutions of a scene can be used with current hardware.

Neural Volumes [LSS+19] is a method relying on this kind of representation. Their method involves an encoder network that encodes input images into a latent representation. This representation is than concatenated with the camera pose of the target view. A decoder network is then applied to predict a voxel based representation (with transparency) of the scene. This representation is afterwards rendered using a differentiable ray marching algorithm. Warp fields, a technique that simulates a dynamic non-uniform grid, is additionally used to increase the resolution of the voxel grid. The method produces high quality results for images of small objects like hands or heads. It also is able to render semi transparent materials such as smoke due to the use of voxel transparency. Perhaps because of the memory limitations mentioned earlier this method was only evaluated on relatively small scenes and it remains questionable whether it is able to scale to large outdoor scenes.

Another voxel based method is Deep Voxels [STH+19]. In this work they design a complex neural network architecture. It first encodes the source image using a 2D U-Net. Lifts the representation to 3D. This 3D "Deep Voxel" representation is than processed by Gated Recurrent Units and subsquently by a 3D U-Net. The result is than projected into the 2D target view, processed by an occlusion network and rendered using a 2D U-Net. The method is again only evaluated on small objects where it showed good quality results using 32^3 voxels. Notably this method reportedly only needs 71 ms to render a single frame.

Point Clouds are a representation that models the 3D world in terms of 3D continuous points (that may have a color attached to it) sampled along surfaces of objects. They are the natural result of Lidar scanners or RGB-D cameras. They can be relatively easily processed by neural networks compared to meshes and they don't waste memory when modeling free space (which happens with naive voxel representations). On the downside they require a large amount of points to represent a scene (compared to meshes) and objects represented by them will have a lack of detail.

Neural Point Based Graphics [AUL19] is a method that renders novel views of a scene given as input raw point clouds. Such point clouds can for example be obtained using a RGB-D camera. These 3D points are augmented with learned Neural Desciptors (vectors like the rgb channels). To render an image the points are rasterized into an images of various size of the target view. These are then fed into different layers of a rendering U-Network which produces the final image. Although not free from artifacts and blurriness their method produces high quality results on single object, persons, indoor rooms and (as shown in later comparisons) also unbounded outdoor scenes. To achieve these results their method requires fine-tuning on the target scene. Notably this methods allows a certain amount of editing the scene by manipulating the underlying point cloud. It also is able to render it's images in only 62 ms (which is not far away from the 30-40 ms required for real time graphics).

Meshes (or polygon meshes) are by far the most popular representation for 3D scenes in traditional computer graphics. They can be seen as a generalization to point

clouds. Rigid objects are represented by a set of points (*vertices*), *edges* that connect them and *faces* that group multiple edges to represent a surface (usually triangles). They optionally can have additional information like textures that describe the colors along a surface. One advantage of meshes is that they can represent flat surfaces with much less data than a point cloud. An example would be for example a flat wall, which can be described by only two triangles whereas a point cloud would need to have lots of points along the wall to represent it's shape. Also the continuity of faces makes it easy to determine which object covers others behind it. When rendering point clouds it often happens that you can see through objects because they are not described continuously. The disadvantage of meshes in novel view synthesis is however that they are hard to deal with for Neural Networks due to their irregular structure [SMG20]. Methods that use meshes as their underlying scene representation therefore often convert meshes into other representations before feeding them into Neural Networks. We still consider those methods to be meshed based methods in this work.

One of the works using meshes as underlying representation is image guided neural object rendering [TZT+18]. In this work they use a multi-view stereo technique to reconstruct proxy geometry of a scene based on a set of input images. Their core idea is that the diffuse part of a color of an object does not change when rendering it from another direction. Therefore they render the diffuse part of the image using traditional image based rendering techniques. On top of that they use a neural network called Effects Net that predicts the view dependent lighting effects. The network first removes the view dependent effects from the input image and then re-applys view dependent effect to the output image. For a novel view this is done based on multiple input images. The resulting multiple output images are then combined into a single one using a composition network. The method was shown to work on scenes containing a small object like a globe. It is reported to run at 50 Hz during test time.

In Deferred Neural Rendering [TZN19] they combine the traditional graphics pipeline with learned components. Specifically they use neural textures to describe the surface of a mesh. This mesh is than rendered using traditional techniques. The rendered image is then translated into the final output image by a neural network converting the screen space feature map to RGB colors. The method is evaluated on scenes containing small objects and shows photo realistic results there. The method reportedly runs in real time.

In Learning Neural Light Transport [SMG20] they address a slightly different problem, where a mesh based 3D representation is given as the input. The mesh is then converted into a point cloud by uniformly sampling points along the surfaces. This point cloud combined with additional properties like albedo is then fed into a neural network processing those points. They are then projected to 2D and occluded points are removed. A Image Synthesis Layer than turns the projected features into an RGB image. Interestingly the method was trained using only noisy monte-carlo renderings

which they show to yield unbiased gradients. The method was evaluated on synthetic scenes of indoor rooms. The method reportedly runs in real times. One thing to note here is since full mesh based 3D representation is given one could also obtain photo realistic images using traditional computer graphic methods such as bi-directional path tracing. These global illumination methods however typically are very slow taking hours instead of ms to create single images. The proposed method runs in real time and it would therefore be interesting to see it compared to traditional real time rasterization methods.

Free View Synthesis [RK20] is a method that takes as input a set of raw images and from that reconstructs a coarse mesh representation and camera poses of the scene using the structure from motion package COLMAP [SF16]. From this mesh representation and camera poses depth maps are rendered for the corresponding images. The method therefore does not rely on RGB-D image to be available like in previous work. To render an image from a new camera pose a depth map is rendered for the new camera pose as well. Additionally the nearest images in the training set are determined by a simple algorithm that transforms all input images into the target view and counts the number of pixel whose depth value lies within 1% of the target depth. The images with the highest counts are then selected as input images. These input images and their depth maps are then each encoded using a U-Net like neural network [RFB15]. Using the depth information from the target view and the camera matrices, the encoded features are then transformed into the target view. A recurrent network then aggregates these features into a confidence and color image. Finally these are combined to the final output via a soft-argmax. The method is evaluated on large unbounded outdoor scenes (from the Tanks and Temples data set [KPZK17]) as well as small objects and indoor scenes. While still being plagued by some artifacts it demonstrates impressive levels of photorealism on such challenging scenes. This improvement comes at a heavy price in computation time (and algorithm complexity) however. Rendering a single image takes on the order of seconds which is too slow for real time graphics (which is the main application area of novel view synthesis). We will investigate this method in more detail later.

Implicit Representations [MON+19, CZ19, PFS+19] describe scenes by functions (Neural Networks) which continuously predict the occupancy probability (and or signed distance function) of every point in the scene.

Neural Radiance Fields (NeRF) [MST+20] is a recent method bringing such implicit representations to the field of novel view synthesis. They revive the classic ideas of viewing the scene as a 5D light field. This light field is approximated by a fully connected neural network that takes in a 5D vector containing a 3D point and a 2D viewing direction and outputs a radiance and density at that point. Classic volume rendering (which is differentiable) is used to render the final image. The positions are additionally encoded using sinus and cosinus functions to help the Neural Network approximate higher frequency functions. To improve sampling efficiency two Neural Networks are used one "coarse" and one "fine". The evaluation of the method

demonstrates remarkable image quality on single objects and bounded scenes (such as rooms). This image quality also comes at a heavy computational price. Rendering a frame takes reportedly 30 seconds and preprocessing/optimization the scene about 1 to 2 days.

A lot of works have quickly followed up to improve/adapt NeRF in various ways [DYC21]. One of them is NeRF++ [ZRSK20] where they analyze the method in more detail. One problem when optimizing a NeRF is what they call the Shape Radiance Ambiguity. In essence the problem is that there are solutions that can perfectly fit the radiance of the training set independently of the underlying geometry. Finding such a solution would lead to catastrophic failure on test time. They claim that this is/can be overcome by regularization. The second point they look at is the parametrization of unbounded scenes. NeRF assumes that the scene fits on an object, which leads to a loss of resolution if the whole scene is modeled or artifacts in the background if only a part of the scene is modeled. They propose to address this problem by having two NeRFs. One that predicts the foreground and one that predicts the background (on an inverted sphere). They show that combining these two predictions leads to an overall higher quality image on unconstrained outdoor scenes. As open challenges to be addressed yet they state rendering speed, robustness and photometric effects.

2.2 Tangential Work

Novel View Synthesis is a field that is closely related or embedded in many other fields such as Computer Graphics, Deep Learning, 3D Reconstruction, Computer Vision or Neural Rendering. In this section we will review some tangential work from these neighboring fields.

A comprehensive State of the Art report on Neural Rendering is available at [TFT+20]. Neural rendering is a field encompassing novel view synthesis and has been defined by the authors as:

"Deep image or video generation approaches that enable explicit or implicit control of scene properties such as illumination, camera parameters, pose, geometry, appearance, and semantic structure."

In the report they give an overview over all the method that exist in that field. They categorize them according to how the neural network is controlled, which classical computer graphics modules are integrated, if the scene is controlled via explicit parameters or implicit examples, how one can control different outcomes of the network and whether the method is scene specific or general. They also outline different application areas of neural rendering such as: Semantic Photo Manipulation, Novel View Synthesis, Free Viewpoint Video Relighting and Facial/Body Reenactment. As main open challenges they state: Generalization, Scalability, Editablity and Multimodal Neural Scene Representations

One fundamental building block of many of the methods described in section 2.1 is the U-Net [RFB15]. It was first introduced as a method for semantic segmentation but it has become clear that one can also use it successfully for other image to image translation tasks. An example would be converting images represented by neural features to RGB. The U-Net is a fully Convolutional Neural Network architecture that reduces the width and height of image layers and increases the number of channels through the use of convolutional layers and max pooling operations. It then subsequentially reverses this process through the use of up-convolution layers. Layers of the same width and height are connected via skip connections to avoid the vanishing gradient problem. The number of channels is changed using 1x1 convolutions. Due to the fully convolutional nature of the architecture it can be applied independently of the images height and width.

As we have seen earlier point clouds are one of the fundamental representations one can use for a Novel View Synthesis method. Processing a point cloud with a neural network however is not trivial as the number of points will be variable in each scene. Point Net [QSMG17] is an neural network architecture that was proposed to address this challenge. It's key features include:

1. Each point is individually processed by an MLP (who all share the same weights).
2. A max pooling layer that aggregates information from all points (if you take the max it doesn't matter how many points you input only 1 value will come out).
3. Depending on the task these global features are than fed into a mlp to output class labels for the whole point cloud or concatenated back to the encoded points to predict labels for every single point.
4. T-Networks are used for feature transformation.

PointNet was originally introduced for point cloud classification and segmentation. It can and is also used as a building block for other methods that process point clouds.

PointNet++ [QYSG17] is an extension to that work that addresses the problem that PointNet does not capture local structure from the metric space. The proposed solution addresses that issue by sampling points into local groups (clustering them if you will) and applying PointNets to these local groups. This process is repeated multiple times in a hierarchical manner until fewer and fewer points remain. For semantic segmentation this process is then reverted and stabilized with skip connections. For classification the final set of points are simply classified with a PointNet. This increases classification and segmentation performance but comes at the cost of decreased forward pass speed.

Two important problems that can come up in novel view synthesis are camera pose estimation and 3D reconstruction from images. The classic structure from motion pipeline is able to address both of these issues [SF16]. The pipeline's goal is to

reconstruct a 3D geometric model and camera poses of a scene given a set of images from that scene. It can roughly be described as follows:
Correspondence Search: In this step we find pairs of images that overlap and create a graph of image projection for each point.

1. Feature Extraction: First we extract features from the images (SIFT, learned, etc.)
2. Matching: We match features between the different images.
3. Geometric Verification: via Homography, Epipolar Geometry or the RANSAC technique.

Incremental Reconstruction: In this second step the image poses are estimated and the scene is reconstructed as a point cloud.

1. Initialization: We start the incremental reconstruction by selecting two images.
2. Image Registration: Add a new image by solving a perspective-n-point problem.
3. Triangulation: From the new image we add new points to the scene
4. Bundle Adjustment: Refine points and camera parameters by minimizing re-projection error
5. Outlier Filtering: Detect and remove outliers
6. Repeat with image registration until you are done.

A popular open source implementation of structure from motion is the package COLMAP [col].

Judging the quality of a generated image with a quantitative metric is a nontrivial problem. Classic per pixel metrics such as mean squared error or Peak Signal to Noise Ratio have the problem that they assume pixel wise independence. A recent development for such metrics/loss functions are the perceptual metrics. They use pretrained convolutional neural networks and cut of later layers. The generated image and the ground truth image are then both fed into this network. The loss of the generated image is then defined as the distance between the activations of the last layer. In [ZIE+18] they show that these perceptual losses outperform classic loss functions such as SSIM or PSNR by a large margin.

As we have argued in the introduction benchmarks are an integral part for the progress of a field in machine learning. The KITTI dataset [GLU12] as the name suggest can be seen as the predecessor of the KITTI-360 dataset we are investigating in this work. It contributed (at that time challenging) benchmarks in the areas of stereo vision, optical flow, SLAM/visual odometry and 3D object detection. It contained data from a laser scanner, cameras as well as GPS.

Fully connected layers, convolutional layers and recurrent layers can be seen as the most basic building blocks in Neural Networks (along with activation functions). Deep Learning recent successes in Computer Vision and Natural Language Processing are the result of powerful convolutional and recurrent architectures respectively. In recent years a new type of fundamental building block has emerged in Natural Language Processing called the self attention. In their seminal paper "Attention is all you need" [VSP+17] the authors showed that it is possible to build natural language processing models based solely on self attention layers (with no recurrent layers). It has since then become the dominant approach in that field. Self Attention layers also have been successfully applied in image classification [ZJK20] where they showed that it can fully replace the convolutional layer and yield similar or better performance. In novel view synthesis the self attention did not yet arrive but it is expected to have an impact on future work by some [tra].

One of the papers first introducing the implicit scene representations we discussed earlier was [MON+19]. Their core idea was to represent 3D objects by continuous decision boundaries that are represented by neural networks. The occupancy network takes as input a 3D point and an observation of an object and predicts an occupancy probability for that point. It is composed of fully connected layers and ResNet blocks. Different types of inputs are handled with different types of encoders such as ResNet, PointNet or a 3D Convolutional architecture.
One downside of plain occupancy networks is that the represented objects have no color or texture. In [OMN+19] they addressed this shortcoming by proposing Texture Fields. Texture Fields are also Neural Networks that take as an input a 3D shape and a single 2D image and learn a function that map 3D points to color values. The texture field architecture is a fully connected ResNet while the image is encoded with a ResNet-18 [HZRS16]. The shape is encoded using a PointNet.
Another limitation of the original implicit representations is that they can only represent small objects. This is likely due to only using simple fully connected layers. The natural extension in a field related to images seems to always be the convolutional layer. In [PNM+20] they introduce an architecture for implicit representation using convolutional layers call Convolutional Occupancy Networks. The input to their method is a point cloud. The point cloud is first encoded using a shallow PointNet. These points are then projected to either a single plane, 3 axis aligned planes or discretized into voxels. Depending on the mode they are then processed by a 2D U-Net, three 2D U-Nets or a 3D U-Net. These features are then aggregated and subsequently processed by a small fully connected occupancy network. This allows the method now represent scenes/objects of the size of rooms.

This concludes the discussion of related and tangential work.

3 Methodology

3.1 The Dataset KITTI-360

The KITTI-360 dataset is a large scale dataset containing information from a variety of different sensors. It's core purpose is autonomous driving. It contains over 320.000 images of a car driving through suburbs of Karlsruhe as well as over 100.000 laser scans. There are semantic class labels available for both the point clouds and the images [XKSG16b]. For the task of Novel View Synthesis the provided images pose a challenging problem since the images are from unconstrained outdoor scene. Due to the recording from a car the images are recorded along a trajectory and rotation of the camera is strictly connected to translation (as a car cannot rotate without moving). Depending on the current driving speed the amount of images available per square meters also is reasonably sparse. We think this dataset can serve as a good benchmark to compare different Novel View Synthesis algorithms on and measure the progress of the field. In this work we will present two baseline algorithms to serve as a starting point for such a benchmark. Firstly in section 3.2 we will discuss a very naive baseline without any machine learning or parameter-tuning to serve as a bottom line for what each more advanced method should at least achieve. Secondly we will investigate a state of the art method in section 3.3 to demonstrate what is already possible and as a target for new methods to beat. Lastly due to the availablity of ground truth semantic segmentation labels one can also wonder if it is possible to synthesize the corresponding semantic labels with the new image as well. We will briefly investigate and discuss this new field in section 3.4. Scenes from the KITTI-360 dataset can contain thousands of images. To keep computation times reasonable we will be restricting our experiments to a subset of 385 images from the first scene.

3.2 Point Cloud Rendering

We call out first baseline Point Cloud Rendering. As the name suggest the idea is to simply render a colored point cloud. Such a point cloud can be obtained in multiple ways. One way would be to use the point clouds from the laser scanners available in the KITTI-360 dataset. Such a point cloud naturally has no colors. To change that one can project and aggregate the colors of the input image pixels onto the points in the point cloud. This is the approach we have used in our experiments. Another way

would be to run a structure from motion package such as COLMAP on the input images and use the resulting dense point cloud.

We render this point cloud through the use of a simple pinhole camera model where one loops through all points and determines the pixel where it will land through dividing by the z component.

```
// convert to image coordinate
int x = static_cast<int>((targetCloud->points[i].x /
    targetCloud->points[i].z) + 0.5f);
int y = static_cast<int>((targetCloud->points[i].y /
    targetCloud->points[i].z) + 0.5f);
```

The points of course will only be rendered if their index lies within the image boundaries and the point is in front of the camera (meaning it's z value is positive). For this to work one needs to first transform the point cloud to the local coordinate system of the desired target view by using the standard intrinsic and extrinsic camera matrices.

```
//Transform point cloud from global space to camera space
PointCloudT *targetCloud = new pcl::PointCloud<PointT>;
PointCloudT *camCloud  = new pcl::PointCloud<PointT>;

// First, project laser point to the current frame
Eigen::Matrix4f invMatrix_p = CameraPoses[(imageIndex -
    startIndex)].inverse();
pcl::transformPointCloud(*coloredPointCloud, *targetCloud, invMatrix_p);

// Second, project cloud from laser coordinate to the camera coordinate
Eigen::Matrix4f invMatrix_e = cameraExtrinsicMatrix[0].inverse();
pcl::transformPointCloud(*targetCloud, *camCloud, invMatrix_e);

// Third, project cloud to the image plane with z
pcl::transformPointCloud(*camCloud, *targetCloud,
    *cameraMatrixPerspectiveIntrinsics);
```

This approach has three problems. First one will be able to see through objects as points do not define a continuous surface. To address this issue will use a proxy mesh (which can be obtained by running a meshing algorithm like poisson on the point cloud) to perform an occlusion check. Using this proxy mesh one can obtain a depth map for the target image. Points whose depth is greater than that of the target mesh will not be rendered as they are considered to be occluded.

```
//Only render points that are projected into the image and in front of the
    plane
if (   x < width
    && x >= 0
```

```
    && y < height
    && y >= 0
    && targetCloud->points[i].z>0)
{
    float intensity = targetCloud->points[i].z;

    //mesh depth check
    float d = -1.0f;
    float depth_threshold = 0.5f; //Need to give a bit of leway if a point
        is right on a mesh
    if (depthCheck)
    {
        d = depthBuffer[y * width + x];
    }

    //Check depth according to mesh. If a point is behind a mesh don't
        render it because it should be occluded
    if (   (d != -1 && ((-d + intensity) < depth_threshold))
        || (d == -1))
    {
        // check depth. If multiple points are rendered on the same pixel
            only use the nearest one
        if (    depth.at<float>(y, x) == 0
            || (depth.at<float>(y, x) != 0 && depth.at<float>(y, x) >
                intensity)
           )
        {
            depth.at<float>(y, x) = intensity;

            targetImage.at<cv::Vec4b>(y, x)[0] = targetCloud->points[i].b;
            targetImage.at<cv::Vec4b>(y, x)[1] = targetCloud->points[i].g;
            targetImage.at<cv::Vec4b>(y, x)[2] = targetCloud->points[i].r;
            targetImage.at<cv::Vec4b>(y, x)[3] = 255; //Set alpha to 100%
                for all rendered pixels
        }
    }
}
```

The second issue is that the resulting image will be rather empty since the number of visible points is usually smaller than the number of pixels we wish to render. To address this issue we fill in the empty pixels by looking up the nearest neighbor in screen space with a valid color. This will ensure that all the pixels will have a valid color value. On the downside this will introduce a unnatural Voronoicell like structure into our resulting image. Naively looking up the nearest neighbor of each pixel has costs in order of O(number of empty pixels * number of filled pixels points). Since images have pixels from in order of hundred thousands this can be somewhat slow. We therefore speed up the lookup by creating a KD-Tree from the rendered

pixels.

```
//Check which which pixels are rendered
for (int x = 0; x < width; x++)
{
    for (int y = 0; y < height; y++)
    {
    ...
    if (NNInterpolation == true)
        {
            if (targetImage.at<cv::Vec4b>(y, x)[3] == 255)
            {
                //Save which points are rendered in image coordinates so
                    that we can do image space NN interpolation later on.
                renderedPointsCloud->push_back(PointT(x, y, 0.0f,
                    targetImage.at<cv::Vec4b>(y, x)[2],
                    targetImage.at<cv::Vec4b>(y, x)[1],
                    targetImage.at<cv::Vec4b>(y, x)[0]));
            }
        }
    }
}

if (NNInterpolation == true)
{
    //Can only do interpolation if we have any points
    if (!renderedPointsCloud->empty())
    {
        //Create KD tree for fast NN search
        kdTree->setInputCloud(renderedPointsCloud);
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                //If there was no point at a pixel the pixel will have 0
                    alpha
                if (targetImage.at<cv::Vec4b>(y, x)[3] != 255)
                {
                    int k = 1;
                    std::vector<float> tmpDistance;
                    pcl::Indices NNIndex;
                    NNIndex.resize(k);
                    //Find the nearest neighbor pixel to fill the color with
                    int numberNeighbors = kdTree->nearestKSearchT(PointT(x,
                        y, 0.0f), k, NNIndex, tmpDistance);

                    if (numberNeighbors > 0)
                    {
                        PointT NN = (*renderedPointsCloud)[NNIndex[0]];
```

```
                    targetImage.at<cv::Vec4b>(y, x)[0] = NN.b;
                    targetImage.at<cv::Vec4b>(y, x)[1] = NN.g;
                    targetImage.at<cv::Vec4b>(y, x)[2] = NN.r;
                    targetImage.at<cv::Vec4b>(y, x)[3] = 255;
                }
            }
        }
    }
}
```

Lastly the sky region will contain no points since the laser scanner cannot detect any objects there. The nearest neighbor algorithm would therefore assign the wrong colors to the sky. To address this we in-paint all pixels in the upper half of the image that do not have a valid depth value in our depth map with a constant sky blue color.

```
cv::Mat mask      = cv::Mat::zeros(height, width, CV_32F);
ProjectMesh2Img(camCloud2, triangles->polygons, sourceImage, depthBuffer,
    *cameraMatrixPerspectiveIntrinsics, meshImg, mask);

if (skyPaintIn == true)
{
    //Sky will never be in the lower half of the image so we hardcode that
        prior.
    cv::Mat lowerHalf = mask(cv::Rect(0, mask.rows / 2, mask.cols,
        mask.rows / 2));
    lowerHalf.setTo(255.);
}
...

//Check which which pixels are rendered
for (int x = 0; x < width; x++)
{
    for (int y = 0; y < height; y++)
    {

        if (skyPaintIn == true)
        {
            //Apply the sky mask. If a pixel is recognized as sky paint it
                blue.
            if (mask.at<float>(y, x) == 0)
            {
                //Magic numbers are the color code for sky blue
                targetImage.at<cv::Vec4b>(y, x)[0] = 235u;
                targetImage.at<cv::Vec4b>(y, x)[1] = 206u;
                targetImage.at<cv::Vec4b>(y, x)[2] = 135u;
```

```
                targetImage.at<cv::Vec4b>(y, x)[3] = 255u;
            }
        }
        ...
    }
}
```

Results of this algorithm will be discussed in section 4.1

3.3 Free View Synthesis

Free View Synthesis (FVS) is a state of the art method for Novel View Synthesis of unconstrained outdoor scenes[1]. It takes as an input a set of images and from that can generate new images of that scene from a new view. A structure from motion algorithm (COLMAP [col]) is run on the input images to reconstruct a proxy geometry mesh of the scene and retrieve the camera positions (and parameters). For each of the camera poses a depth map will be rendered using the pyrender library (a cython version of librender) [pyR].

```
Rs = np.load(args.camera + "/Rs.npy").astype(np.float64)
ts = np.load(args.camera + "/ts.npy").astype(np.float64)
ts = np.expand_dims(ts, axis=2)

input_file = args.ply
ply_orig = o3d.io.read_triangle_mesh(input_file)

for idx, elem in enumerate(Ks):
    #Apply transformation to current view:
    ply = copy.deepcopy(ply_orig)

    # Convert open3d format to numpy array
    vertices = np.asarray(ply.vertices)
    faces    = np.asarray(ply.triangles)

    #Rotate and translate points
    vertices = (Rs[idx] @ vertices.T)
    vertices = (ts[idx] + vertices).T
    #Due to the transpose the arrays are Fortran Contiguous. We need to
        convert them back to C contiguous
    vertices = vertices.copy(order='C')

    # Get height and widths of the images.
    source_img = Image.open(args.input_image + "/" + args.image)
    heigth = source_img.height
    width = source_img.width
```

[1]Some unpublished methods that achieve higher quality results exist on archive already

```
    img_size = np.array((heigth, width))

    fx = Ks[idx,0,0]
    fy = Ks[idx,1,1]
    px = Ks[idx,0,2]
    py = Ks[idx,1,2]

    cam_intr = np.array((fx,fy,px,py))

    # vertices: a Nx3 double numpy array
    # faces: a Nx3 int array (indices of vertices array)
    # cam_intr: (fx, fy, px, py) double vector
    # img_size: (height, width) int vector
    depth_map = pyrender.render(vertices, faces, cam_intr, img_size)
```

The depth maps are used together with the camera poses in a simple source selection algorithm that determines for a target view the nearest (approximately) images in the input set. The algorithm projects every pixel of the depth map (with a valid depth value) into the domain of all source images (respectively). It then counts the number of pixels where the target depth value lies within 1% of the source depth value. The images with the highest counts will then be chosen later on as input images for the synthesized view. Below is a vectorized python implementation of the algorithm. Note that this is still very slow (takes about 5 hours for roughly 400 images). For that reason we use a Cython function in our implementation to speed things up further (to roughly 90 minutes).

```
for idx in tqdm(range(0, num_img)):

   #Cython function that does the same as below but faster
    #counts = preprocess.count_nbs(depthMaps[idx], source_Ks[idx],
        source_Rs[idx], source_ts[idx], depthMaps, \
    #                            source_Ks, source_Rs, source_ts, 0.01)

    #Check overlap of every pixel in current file
    for y in tqdm(range(0, heigt)):
        for x in range(0, width): #Inside takes rougly 0.06ms
            #Check for valid depth value
            z = depthMaps[idx,y,x]

            if(z <= (0.0 + numerical_tolerance)):#Skip if depth value is
                invalid
                continue

            point_tgt = np.array([[x * z],[y * z],[z]], dtype=np.float64)
                #Undo pinhole camera

            #Transform to global coordinate system.
            point_global  =  np.linalg.solve(source_Ks[idx], point_tgt)
```

```
            #Apply inverted matrix by solving linear system
        point_global   =  source_Rs[idx].T @ point_global #Inverse
            Rotation
        point_global   = (-(source_Rs[idx].T @ source_ts[idx])) +
            point_global #Inverse Translation

        #Transform to source coordinate system
        point_src = source_Rs @ point_global
        point_src = source_ts + point_src
        point_src = source_Ks @ point_src

        #Pinhole camera model
        src_x = point_src[:,0,0] / point_src[:,2,0]
        src_y = point_src[:,1,0] / point_src[:,2,0]

        # If our current pixel is visible in this source image increase
            the count
        cond1 = np.greater_equal(src_x, zeroArray)
        cond2 = np.less(src_x, widthArray)
        cond3 = np.greater_equal(src_y, zeroArray)
        cond4 = np.less(src_y, heightArray)

        tmp = np.logical_and(cond1, cond2)
        tmp = np.logical_and(tmp, cond3)
        tmp = np.logical_and(tmp, cond4)

        #depth check with sourcedepth map in 1% range
        #Avoid illegal indices
        src_x[np.logical_not(tmp)] = 0
        src_y[np.logical_not(tmp)] = 0

        #Round down float values to pixel indices
        idX2 = np.floor(src_x + numToleranceArray).astype(np.int)
        idY2 = np.floor(src_y + numToleranceArray).astype(np.int)
        projected_target_depth = np.ones(num_img) * point_src[:,2,0]

        valid_depth_greater =
            np.greater(depthMapsUpper[np.arange(num_img),idY2,idX2],
            projected_target_depth)
        valid_depth_smaller =
            np.greater(projected_target_depth,depthMapsLower[np.arange(num_img),idY2,idX2]

        tmp = np.logical_and(tmp, valid_depth_greater)
        tmp = np.logical_and(tmp, valid_depth_smaller)
        counts[tmp] += 1

    counts[idx] = 0 #Set your own count to 0 to avoid using yourself as
        input
```

The input images, counts, depth maps and camera matrices are then processed by a neural network to generate the new views. The architecture as discussed earlier encoded the K nearest input images and depth maps using U-Networks, transforms them into the target view and then combines them using a block based on convolutional gated recurrent units (GRU). It outputs per pixel confidence values and a color image which is then finally combined to the output image using a soft-argmax. For the blending network we are using the official github implementation [Frea]. Retraining the Neural Network takes reportedly 12 days [Freb] which is why we are using the provided pretrained weights without any parameter-tuning (which works surprisingly well). The reported results are therefore zero shot results.

Converting the COLMAP output to the Neural Network input takes some more technical steps (such as converting file formats) which we will skip here because they are conceptually uninteresting. Two things will be noted however.
First COLMAP is an algorithm implemented in parallel. Hence the resulting camera poses are not necessarily saved in order (consecutively). Since we need to match cameras with images and depth maps one needs to reorder them based on the image names.
Second when doing a quantitative evaluation of the method one needs to be careful with the COLMAP process. Naively running COLMAP on all your images will lead to a dataleak since the proxy geometry will be of higher quality having seen additionally the evaluation images therefore biasing the results. Simply running the COLMAP twice once for the input and once for the evaluation images also does not work because the resulting camera poses and proxy geometry will not lie in the same global coordinate system. We addressed this issue by first running COLMAP on all the images to obtain the camera poses. Then subsequently we do a reconstruction from known camera poses [Rec] only on the input images. This is a nontrivial process involving database and text editing/sorting which we automated through some python scripts. Our evaluation results can be seen in section 4.2.

3.4 Novel Label Synthesis

Relatively accurate synthesis of novel views has been demonstrated in the past. Therefore allowing the generation of an (almost) arbitrary amount of raw images. One might wonder if it is also possible to synthesize high quality labels alongside those images. Obtaining raw images them-self's is relatively cheap (and therefore the main goal of view synthesis should real time generation of those images) but creating labels for them (especially semantic segmentation labels) requires a lot of manual labor and therefore is time and cost intensive. Being able to freely synthesize labels would dramatically increase the amount of labeled image data that is available to train a model. For example neural networks are usually trained in epochs where one iterates over the same data multiple time biasing the algorithm to over-fit to

that set of data. Being able to synthesize new images along side labels would enable practitioners to only train on each sample once even if the original labeled dataset is small. One could view such a method as an advanced form of data augmentation.

There are two possible ways on how to define such a task.

- Given a set of images *with their respective labels* synthesize new labeled images of that scene.
- Given a set of images synthesize new labeled images of that scene.

The second definition is clearly more general and would allow for labeling and expanding unlabeled datasets. Without designing any new method one can already pursue such an approach by first creating a labeled image using a Novel View Synthesis method (like FVS) and than subsequently label the generated images using a semantic segmentation network. We will demonstrate this with a short experiment in section 4.3 where we synthesize images using free view synthesis and label them using a pre-trained PSPNet.
However this problem definition does in a way represent a chicken or the egg problem. Labeled data are used for training labeling methods but to generate the labeled data one already needs a labeling method. It is unclear whether a method trained on generated labels is not bound by the performance of the method generating these labels (put it in other words a method imitating an expert could be bound by the performance of that expert). One can of course combine both steps synthesizing the image and labeling it into one model that does both. This can speed up computation times as the same intermediate representations can be reused for both task but it remains an open research question if combining these two steps can yield any improvement in accuracy.

The first approach by definition is simpler but only allows for extension of labeled datasets. We think however that it does circumvent the aforementioned problems because the task of synthesizing new labels from given once is fundamentally different from labeling and image. To demonstrate that we conduct an experiment where we synthesize new labels (semantic segmentation labels color coded as rgb images) using the pre-trained blending network from Free View Synthesis. The results can be seen in section 4.3. It is able to synthesize labels of reasonable quality without ever having seen a label during training.

4 Experiments

4.1 Point Cloud Rendering

For the Point Cloud Rendering method we show qualitative results and discuss the effects of the various design decisions we made. In figure 4.1 you can see an example where we simply render the colored point cloud without any additional steps. One can observe that in the center where we are driving towards there are enough points to yield a high quality image. Farther to the sides the amount of points is not sufficient anymore to cover all the pixels. The cars on the right are hard to spot and on the left you can see the houses far away through the vegetation. The sky region does not posses any points and is therefore empty. One can also spot the scan pattern on which the points where created.

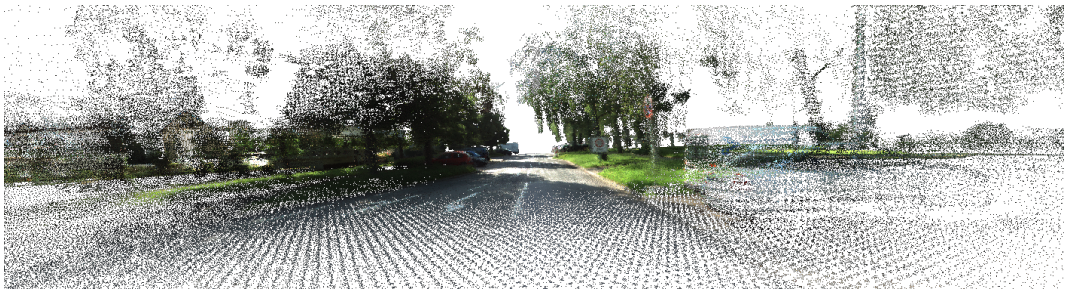

Figure 4.1: Novel View generated by only rendering the colored point cloud

In the next figure 4.2 we show how the nearest neighbor interpolation addresses the issue with the empty pixels. In the center where we have many points the scene looks now very natural. However several problems remain. On can see through objects like the vegetation on the left or the traffic sign on the right. In the sky regions with no points one can clearly see the voronoi patterns.

To address the problem of looking through objects we now add a depth check to our method only rendering points that are on or in front of a proxy mesh. The results can be seen in figure 4.3. The traffic sign on the right as well as the cars on the left can now be seen more clearly. The trees now correctly cover the house behind them. Since the proxy mesh is not perfect it can sometimes occur that too many points are blocked because the proxy mesh is larger than the actual object. One can observe that on the right with the traffic sign. For an illustration of the underlying proxy

Figure 4.2: Novel View generated using Nearest Neighbor interpolation

Figure 4.3: Novel View generated by additionally applying an occlusion check

geometry see figure 4.4 [1].

Finally we will inpaint pixels with invalid depth values in the top half of the image with constant skyblue color. The result of the final algorithm can be seen in figure 4.5.

For another example and a side by side comparison with the corresponding ground truth image see figure 4.6.

One note of caution. The colors of the point cloud were created using all input images. Since we are rendering the same camera poses of those input images the results can

[1] The colors refer to labels applied to the mesh. This information is not used in our method

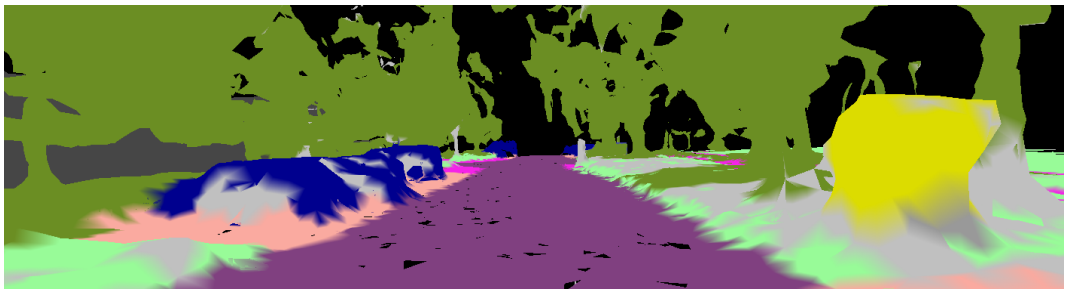

Figure 4.4: Proxy Geometry used in the Point Cloud Rendering algorithm

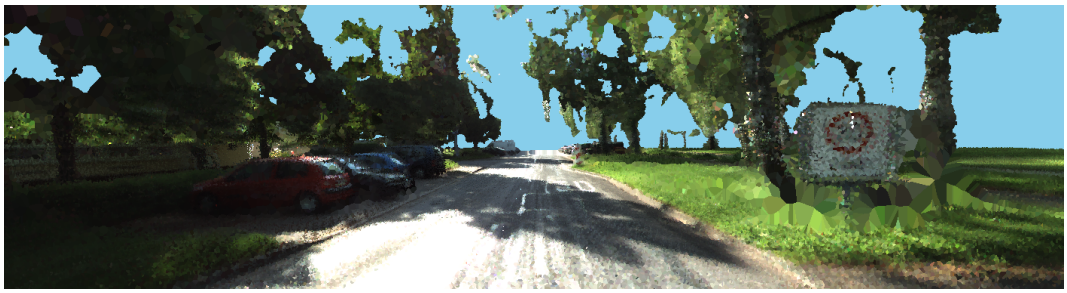

Figure 4.5: Full Point Cloud Rendering algorithm with sky in-painting

Figure 4.6: Side by Side comparison of an image generated with point cloud rendering (bottom) and the ground truth (top)

be said to be biased. Since there are no learned components or parameters in this method we think the difference is negligible.

For a quantitative evaluation of the Point Cloud Rendering method see table 4.1. As one would expect it gets outperformed by the advanced method Free View Synthesis by a large margin on all metrics.

4.2 Free View Synthesis

The number of input images available to a method can determine how difficult a Novel View Synthesis problem is (ideally normalized by the physical size of the scene). For that reason we look at 2 input/eval splits in our investigation of free view synthesis. Firstly we use 50% of the available images (385 total) as input for the method and secondly we drastically reduce that number to only 10% of the available input (and respectively we will have 50% and 90% evaluation images). We perform both a quantitative as well as a qualitative evaluation of the method. The COLMAP algorithm undistorts the images given to him (in our case the image get a few pixels smaller but look exactly the same). We treat the undistorted images as ground truth in our evaluation. For our evaluation we use the official pretrained model and do not perform any training. The results can hence be seen as zero-shot results. The input images are chosen uniformly from all images such that every second image along the trajectory of the car is an input image. In figure 4.7 you can see two qualitative examples of images synthesized by Free View Synthesis. Both images are of remarkably high quality to a point where they are almost indistinguishable from real images. On the top image one can observe that part of the traffic sign and one of the trees is missing. The resulting image sees right through the point where the object should have been. We think this is due to the proxy geometry being too coarse at these points.

We show the corresponding depth maps in figure 4.8. They are relatively coarse but most of the time that seems to be good enough for the blending network to do its job.

Next we drastically reduce the number of input images down to 10%. Two resulting images can be seen in figure 4.9. We observe a big decrease in image quality. A lot of brown artifact appear on the sides. The traffic sign is even more transparent and the tree next to it is completely missing. On the bottom image the quality is even worse where the car seems to be deformed and on the left appear colors of an cigarette-automat that should have appeared earlier in the scene.

We think the reason for this decline in quality is twofold. Firstly the distance to the nearest input image has drastically increased. When looking at the full trajectory one can observe that the image quality increase the closer you get to an input image and decreases when you are moving away from one. The second reason is that the COLMAP reconstruction becomes extremely coarse to a point where most objects are

Figure 4.7: Images synthesized by Free View Synthesis using 50% of images as input

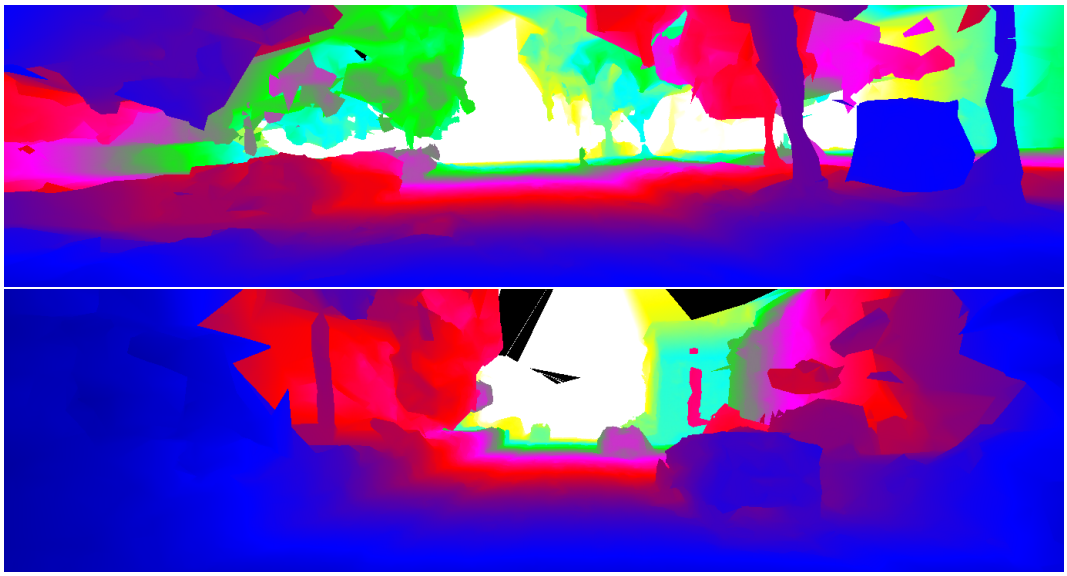

Figure 4.8: Depth Maps used to synthesize the images (50% of images used as input for their reconstruction)

Figure 4.9: Images synthesized by Free View Synthesis using 10% of images as input

Method	↓ L1Distance	↓PSNR	↓ SSIM	↓ LPIPS
Point Cloud Rendering (100%) (385)	0.38401	-13.30166	-0.61063	0.5914
Free View Synthesis 50% (192)	**0.10196**	**-24.71190**	**-0.89850**	**0.1692**
Free View Synthesis 10% (39)	0.21996	-18.73886	-0.77087	0.3366

Table 4.1: Quantitative Evaluation of the Free View Synthesis method (Zero Shot) and Point Cloud Rendering.

missing from the mesh and only the tunnel like shape of the street is preserved. You can observe this in the depth maps shown in figure 4.10. Only very little structure of the scene is preserved.

We also evaluate the method quantitatively using the metrics L1 Distance, PSNR, SSIM and LPIPS. The average over all 385 images is reported. For the LPIPS metric we use the VGG architecture and version 0.1. When looking at the individual numbers we observed that the numbers were slightly better at the beginning of the scene where the car was driving slower and more input images were available (per distance traveled). The results are presented in table 4.1. Notably L1-Distance and LPIPS show a stronger percentage decline than PSNR and SSIM.

Notably the numbers we report for 50% are somewhat similar to the numbers reported in [RK20] for zero shot results on Tanks and Temples. The Free View Synthesis method is therefore reproducible in our setting.

A side by side comparison of a generated image (50%) and ground truth is presented in figure 4.11. We can observe that there is a sort of brown noise in the generated

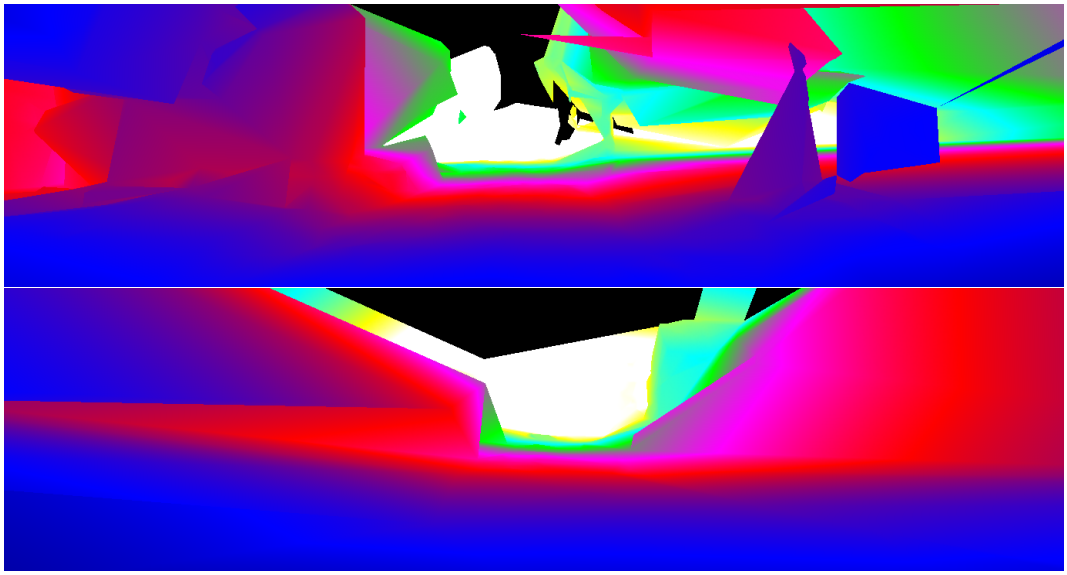

Figure 4.10: Depth Maps used to synthesize the images (10% of images used as input for their reconstruction)

image that is stronger at the sides than in the center. Also a window on the right side appears to be stretched.

4.3 Free Label Synthesis

To investigate the first approach we conduct an experiment where we use the pretrained neural network from Free View Synthesis (alongside the depth maps from the RGB images). Instead of natural RGB images we feed the network labels encoded as an RGB image as an input. The result will of course be an RGB image and not a label. However one can easily turn such a result into a label by projecting the ouput colors to the nearest respective color with a label associated to it. This is of course just a simple proof of concept and not a proper label synthesis method. A result can be seen in figure 4.12. There are multiple things to observe here. First the method as said before tries to produces an image that is not bound to the colors with labels. Therefore we can see some artifacts in the image with invalid colors. Second the overall quality of the labels is remarkably good. Most of the areas have the correct color associated to them and the shape seems to be approximately right. This again demonstrates the robustness of the Blending Network which was only trained on natural images yet still works on these painting like labels. Third the network seems to have a problem with dark blue color associated to the car label. It consistently maps this blue to black across all the images.

Overall we want to show with this experiment that the first task description (label

Figure 4.11: Side by Side comparison of an image generated with Free View Synthesis (bottom) and the ground truth (top)

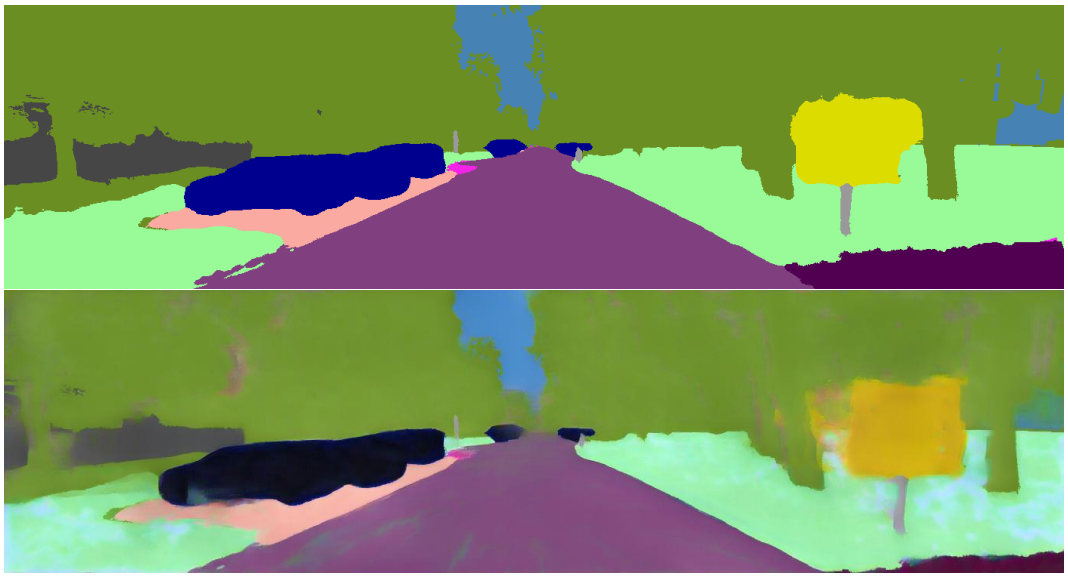

Figure 4.12: A Semantic segmentation (bottom) created by free view synthesis (a method never trained for the task of semantic segmentation). At the top you can see the ground truth data.

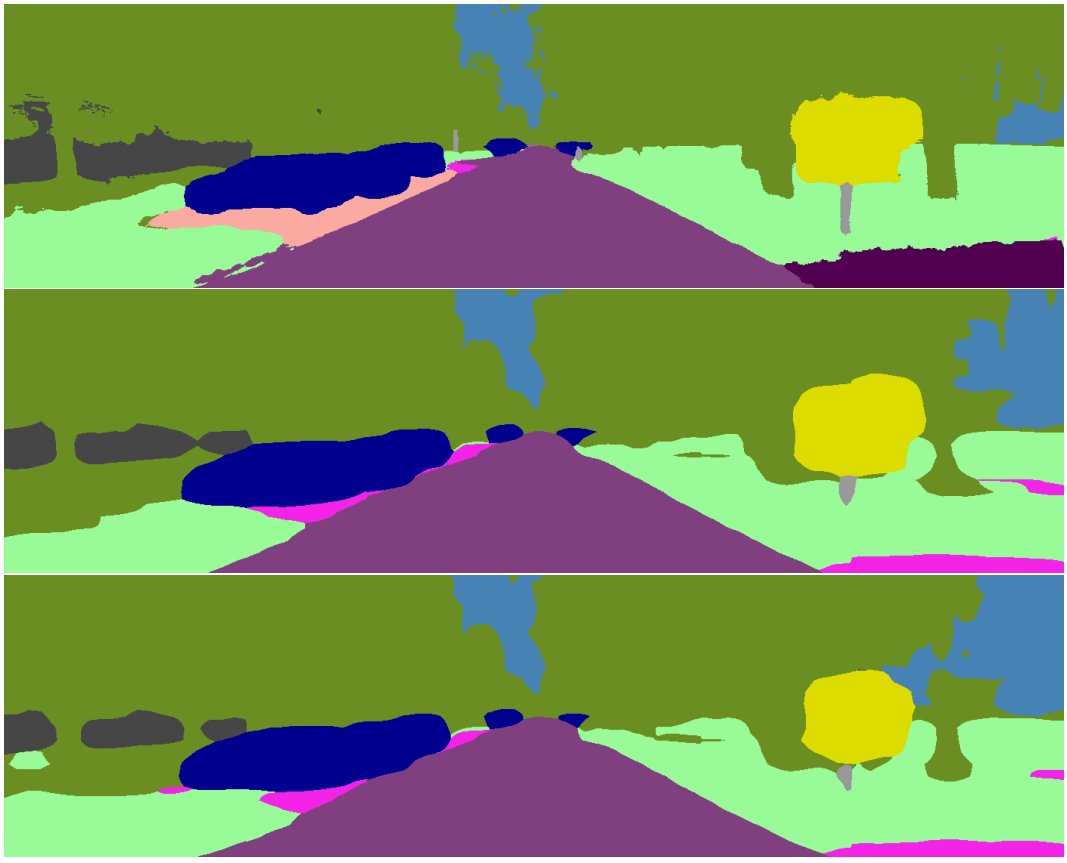

Figure 4.13: A Semantic segmentation (bottom) created by a pre-trained PSP-Net labeling a synthesized image (FVS 50/50). At the top you can see the ground truth data. In the middle you can see the PSP-Net labeling for the real image.

synthesis) is distinctly different from semantic segmentation. After all the blending network has never seen a label or even understands the concept of classes. Yet it is still able to produce reasonable labels. Because this is a distinct task the quality of the labels is not bounded by the strength of available segmentation networks but rather by the strength of synthesizing methods (keep in mind that we want the labels to serve as expert demonstrations to train segmentation networks).

To demonstrate the second problem definition we conduct a second experiment where we label a set of images synthesized by Free View Synthesis using a pre-trained PSP-Network. An example result can be seen in figure 4.13. There does not seem to be a major loss in quality between the prediction on the real image in the middle and the prediction on the synthesized image on the bottom. The PSP-Net labels generally seem to be smoother than the noisy ground truth but also have a lack of detail. For example the wheels of the cars are not as clearly visible and half of the

pole from the traffic sign is missing. On the right there is too much area labeled as sky. One important thing to note is that the PSP-Net we used here was trained on only 19 classes which is a subset of the total available classes in KITTI-360. For example the network does not know the parking class that can be seen in figure 4.13. For the quantitative evaluation we mapped these classes to the most similar once.

A quantitative evaluation can be found in table 4.2. We report the intersection over union (IoU) and accuracy (Acc) metrics both in mean and for each individual class. Because 7 of the classes do not occur in our scene their values are 0. We therefore also report the average values over only the 12 classes that do show up in the images (valid). Unsurprisingly the labels for the real image have almost always the highest values. The images synthesized by Free View Synthesis (50%) on average retain 89.0% of the IoU and 90.6% of the accuracy performance.

A comparison of both methods can be seen in figure 4.14. The middle image has some blur but overall the shapes are fine. The position of the pedestrian seems to be a bit off. The bottom image has relatively smooth labels at the cost of details. That the pedestrian in the bottom image only shows up as a small area the size of a head is concerning as this is the label that is most important not to miss.

A potential limitation we have to keep in mind is that we do not know which scenes from KITTI-360 the PSP-Net was trained on so the training set could have included the scene we synthesized and labeled here.

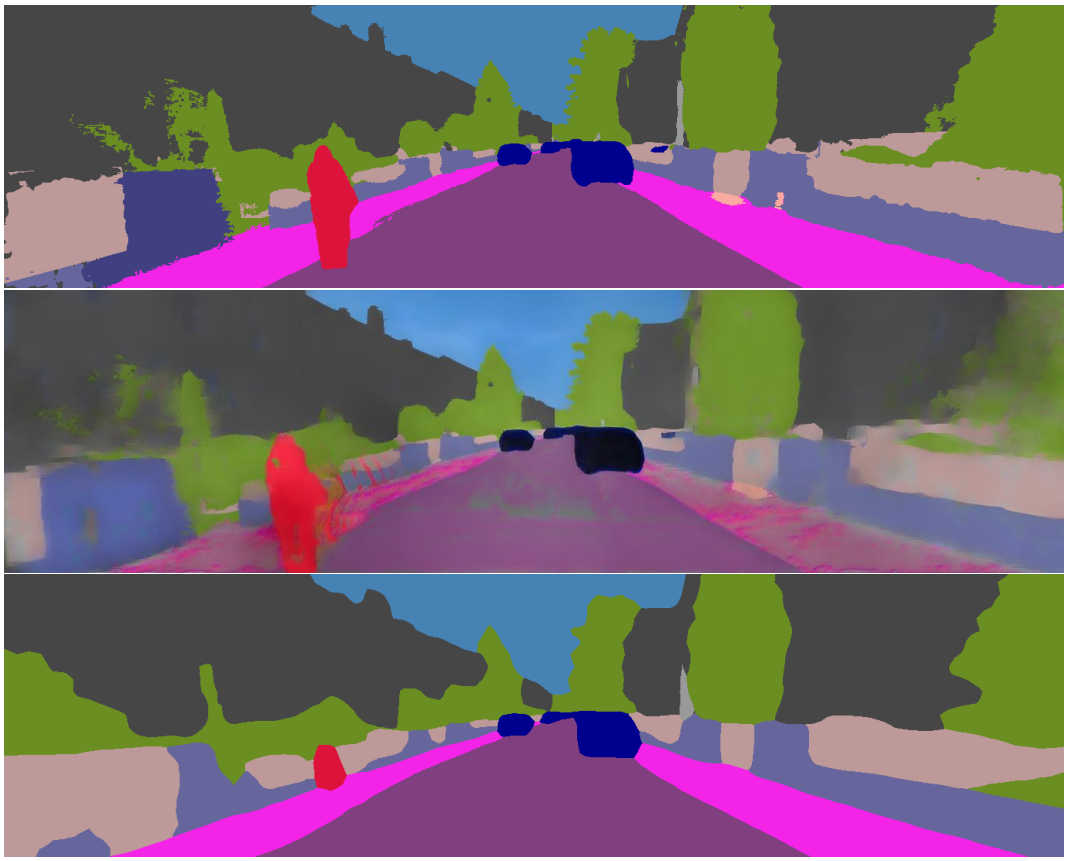

Figure 4.14: A comparison between both methods. Ground truth is at the top. In the middle is the image synthesized with Free View Synthesis. On the bottom is the image synthesized by the PSP-Net.

Metric/Method	PSP-Net Real Image	PSP-Net FVS(50/50)	PSP-Net FVS (10/90)
mIoU/mAcc/allAcc	0.4371/0.4903/0.9047	0.3892/0.4443/0.8878	0.2870/0.3479/0.8038
mIoU/mAcc (valid)	0.6921/0.7763	0.6162/0.7035	0.4544/0.5508
Road iou/accuracy	0.8762/0.8936	0.8661/0.8844	0.8456/0.8836
Sidewalk iou/accuracy	0.7172/0.8693	0.7017/0.8711	0.5805/0.8398
Building iou/accuracy	0.9009/0.9485	0.8616/0.9464	0.6771/0.9239
Wall iou/accuracy	0.6371/0.7880	0.5535/0.7082	0.2493/0.3453
Fence iou/accuracy	0.4980/0.7610	0.4538/0.6888	0.2011/0.2582
Pole iou/accuracy	0.1589/0.1758	0.0437/0.0464	0.0312/0.0322
Traffic light iou/accuracy	0.0000/0.0000	0.0000/0.0000	0.0000/0.0000
Traffic sign iou/accuracy	0.6871/0.7433	0.4357/0.4592	0.1460/0.1473
Vegetation iou/accuracy	0.8675/0.9481	0.8444/0.9261	0.7301/0.8176
Terrain iou/accuracy	0.6864/0.7493	0.6891/0.7553	0.6130/0.7207
Sky iou/accuracy	0.8417/0.9293	0.8094/0.9248	0.7365/0.9207
Person iou/accuracy	0.5625/0.5714	0.2908/0.2938	0.0523/0.0545
Rider iou/accuracy	0.0000/0.0000	0.0000/0.0000	0.0000/0.0000
Car iou/accuracy	0.8720/0.9384	0.8460/0.9363	0.5896/0.6655
Truck iou/accuracy	0.0000/0.0000	0.0000/0.0000	0.0000/0.0000
Bus iou/accuracy	0.0000/0.0000	0.0000/0.0000	0.0000/0.0000
Train iou/accuracy	0.0000/0.0000	0.0000/0.0000	0.0000/0.0000
Motorcycle iou/accuracy	0.0000/0.0000	0.0000/0.0000	0.0000/0.0000
Bicycle iou/accuracy	0.0000/0.0000	0.0000/0.0000	0.0000/0.0000

Table 4.2: Quantitative Evaluation of the PSP-Net labels on the different kind of images (real/synthesized). First we report the mean over all classes. Then the mean only over the classes that occur in the images. Finally we show the metrics for all individual classes.

5 Conclusion

Comparing different Novel View Synthesis methods just based on their papers is difficult because authors usually evaluate on a wide variety of different scenes that are not of the same difficulty. To address this we have proposed to set up the KITTI-360 dataset as a common task framework for the field of Novel View Synthesis. To that end we reproduced and evaluated two methods, Point Cloud Rendering and Free View Synthesis, on the dataset to serve as baselines. We have also discussed and briefly investigated the new field of Novel Label Synthesis that might become possible with the rise of photo-realistic Novel View Synthesis methods. Before concluding we want to make one additional point. We think it is important for a field to keep in mind the underlying task they try to solve. Acquiring photo-realistic images and trajectories of a static real scene in and of itself is not a hard task. One can simply take a camera and record the scene. What you can not do with a plain camera is **interactively** exploring a photorealistic scene. Traditional computer graphics methods also struggle with this scenario and require time and cost expensive modeling of the underlying scene. This is where we think the core task and strength of Novel View Synthesis lies. Generating photorealstic novel views of a scene in **real time**. In Computer Graphics real time usually means 30 Frames per second[1] or more since around that point humans start to experience individual images as continuous. Recent methods such as Neural Radiance Fields or Free View Synthesis have demonstrated a remarkable level of image quality. However this came at a huge computational price, 30 seconds per frame for NeRF and 3-4 seconds per frame for Free View Synthesis [2]. This is too slow as that clever engineering or a faster hardware could fix the problem. A central open question in the field is therefore how to maintain that level of photorealism while gaining several orders of magnitude in speed.

[1] 33 ms per image

[2] That is the time the blending network took in our experiment to generate an image. One needs to add on top of that the source selection algorithm and rendering the depth map.

Bibliography

[AUL19] Kara-Ali Aliev, Dmitry Ulyanov, and Victor S. Lempitsky. Neural point-based graphics. *CoRR*, abs/1906.08240, 2019.

[col] Colmap repository. `https://github.com/colmap/colmap`. Accessed: 2020-01-24.

[CZ19] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 5939–5948. Computer Vision Foundation / IEEE, 2019.

[Don17] David Donoho. 50 years of data science. *Journal of Computational and Graphical Statistics*, 26(4):745–766, 2017.

[DYC21] Frank Dellaert and Lin Yen-Chen. Neural volume rendering: Nerf and beyond, 2021.

[FBD+19] John Flynn, Michael Broxton, Paul E. Debevec, Matthew DuVall, Graham Fyffe, Ryan S. Overbeck, Noah Snavely, and Richard Tucker. Deepview: View synthesis with learned gradient descent. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 2367–2376. Computer Vision Foundation / IEEE, 2019.

[Frea] Free view synthesis github repository. `https://github.com/intel-isl/FreeViewSynthesis`. Accessed: 2020-01-26.

[Freb] Free view synthesis github repository reported training time. `https://github.com/intel-isl/FreeViewSynthesis/issues/2`. Accessed: 2020-01-26.

[GGSC96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In John Fujii, editor, *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1996, New Orleans, LA, USA, August 4-9, 1996*, pages 43–54. ACM, 1996.

[GLU12] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the KITTI vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence,*

RI, USA, June 16-21, 2012, pages 3354–3361. IEEE Computer Society, 2012.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[kit] Kitti-360 website. `http://www.cvlibs.net/datasets/kitti-360/`. Accessed: 2020-01-20.

[KPZK17] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics*, 36(4), 2017.

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. In John Fujii, editor, *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1996, New Orleans, LA, USA, August 4-9, 1996*, pages 31–42. ACM, 1996.

[LSS+19] Stephen Lombardi, Tomas Simon, Jason M. Saragih, Gabriel Schwartz, Andreas M. Lehrmann, and Yaser Sheikh. Neural volumes: learning dynamic renderable volumes from images. *ACM Trans. Graph.*, 38(4):65:1–65:14, 2019.

[MON+19] Lars M. Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 4460–4470. Computer Vision Foundation / IEEE, 2019.

[MSC+19] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion: practical view synthesis with prescriptive sampling guidelines. *ACM Trans. Graph.*, 38(4):29:1–29:14, 2019.

[MST+20] 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. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part I*, volume 12346 of *Lecture Notes in Computer Science*, pages 405–421. Springer, 2020.

[OMN+19] Michael Oechsle, Lars M. Mescheder, Michael Niemeyer, Thilo Strauss, and Andreas Geiger. Texture fields: Learning texture representations in function space. In *2019 IEEE/CVF International Conference on Com-*

puter Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019, pages 4530–4539. IEEE, 2019.

[PFS+19] Jeong Joon Park, Peter Florence, Julian Straub, Richard A. Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 165–174. Computer Vision Foundation / IEEE, 2019.

[PNM+20] Songyou Peng, Michael Niemeyer, Lars M. Mescheder, Marc Pollefeys, and Andreas Geiger. Convolutional occupancy networks. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part III*, volume 12348 of *Lecture Notes in Computer Science*, pages 523–540. Springer, 2020.

[pyR] Pyrender github repository. `https://github.com/griegler/pyrender`. Accessed: 2020-01-26.

[QSMG17] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 77–85. IEEE Computer Society, 2017.

[QYSG17] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5099–5108, 2017.

[Rec] Colmap reconstruction from known camera poses. `https://colmap.github.io/faq.html#reconstruct-sparse-dense-model-from-known-camera-poses`. Accessed: 2020-01-26.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells III, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III*, volume 9351 of *Lecture Notes in Computer Science*, pages 234–241. Springer, 2015.

[RK20] Gernot Riegler and Vladlen Koltun. Free view synthesis. In Andrea

Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XIX*, volume 12364 of *Lecture Notes in Computer Science*, pages 623–640. Springer, 2020.

[SF16] Johannes L. Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 4104–4113. IEEE Computer Society, 2016.

[SMG20] Paul Sanzenbacher, Lars M. Mescheder, and Andreas Geiger. Learning neural light transport. *CoRR*, abs/2006.03427, 2020.

[STH+19] Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Niessner, Gordon Wetzstein, and Michael Zollhofer. Deepvoxels: Learning persistent 3d feature embeddings. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[TFT+20] Ayush Tewari, Ohad Fried, Justus Thies, Vincent Sitzmann, Stephen Lombardi, Kalyan Sunkavalli, Ricardo Martin-Brualla, Tomas Simon, Jason M. Saragih, Matthias Nießner, Rohit Pandey, Sean Ryan Fanello, Gordon Wetzstein, Jun-Yan Zhu, Christian Theobalt, Maneesh Agrawala, Eli Shechtman, Dan B. Goldman, and Michael Zollhöfer. State of the art on neural rendering. *Comput. Graph. Forum*, 39(2):701–727, 2020.

[tra] Who will be the first to combine nerf and transformer. `https://twitter.com/fdellaert/status/1349771661785112576?s=20`. Accessed: 2020-01-24.

[TZN19] Justus Thies, Michael Zollhöfer, and Matthias Nießner. Deferred neural rendering: image synthesis using neural textures. *ACM Trans. Graph.*, 38(4):66:1–66:12, 2019.

[TZT+18] Justus Thies, Michael Zollhöfer, Christian Theobalt, Marc Stamminger, and Matthias Nießner. IGNOR: image-guided neural object rendering. *CoRR*, abs/1811.10720, 2018.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[XKSG16a] Jun Xie, Martin Kiefel, Ming-Ting Sun, and Andreas Geiger. Semantic instance annotation of street scenes by 3d to 2d label transfer. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[XKSG16b] Jun Xie, Martin Kiefel, Ming-Ting Sun, and Andreas Geiger. Semantic instance annotation of street scenes by 3d to 2d label transfer. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 3688–3697. IEEE Computer Society, 2016.

[ZIE+18] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 586–595. IEEE Computer Society, 2018.

[ZJK20] Hengshuang Zhao, Jiaya Jia, and Vladlen Koltun. Exploring self-attention for image recognition. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 10073–10082. IEEE, 2020.

[ZRSK20] Kai Zhang, Gernot Riegler, Noah Snavely, and Vladlen Koltun. Nerf++: Analyzing and improving neural radiance fields. *CoRR*, abs/2010.07492, 2020.

[ZTF+18] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification: learning view synthesis using multiplane images. *ACM Trans. Graph.*, 37(4):65:1–65:12, 2018.

[ZTS+16] Tinghui Zhou, Shubham Tulsiani, Weilun Sun, Jitendra Malik, and Alexei A. Efros. View synthesis by appearance flow. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 286–301. Springer, 2016.