Department of Information Engineering and Computer Science

Master's Degree in

Information and Communication Engineering

Final Dissertation

View Synthesis and 3D Mesh Generation through Neural Radiance Fields

The application of NeRF and Plenoxels to Hand-Drawn Characters

Supervisor
Nicola Conci

Co-Supervisor
Zeno Sambugaro

Student
Andrea Masciulli

Academic year 2020/2021

To my parents,
to whom I owe a lot.

Contents

Abstract

Character turnarounds are pieces of art where a character, imagined by the artist, is drawn multiple times as seen from different directions, while maintaining its pose and volume consistent. Turnarounds find great use in animation studios, since they define a character design and enable drawing consistency among different artists. Such drawings, with the ever-growing progress in computer graphics and view-synthesis research, could be turned into datasets for 3D reconstruction or similar tasks.

This thesis presents a novel use case for view-synthesis SoA techniques - such as Neural Radiance Fields and Plenoxel - which are used to obtain a 3D representation of hand drawn characters out of their 2D character turnaround. Ultimately, we want to be able to render the turnaround's character from unseen views, and even be able to turn the 3D representation into a 3D mesh of the character. Instead of working with images captured by a camera in the real world, the algorithms will have to deal with 2D drawings of a character, whose 3D representation exist just in the artist's mind. This comes with some significant challenges with respect to the standard use case of these view-synthesis algorithms, given the much more limited number of views and unknown camera positions.

Applying view-synthesis techniques to character turnarounds may find interesting applications in areas such as digital animation or 3D modeling, serving as a valuable tool for artists to speed up their drawing and modeling processes.

1 Introduction

1.1 Context and Motivation

Whether an artist draws a character that only they will use, or especially if they create a character that will be handed off to someone else, it is usually not enough to create a single drawing from a single view. Creating turnarounds (or character model sheets) is a great way to establish things about the character thoroughly and figure out ambiguous things about them before they're put to use. In short, turnarounds consist in drawing a character from multiple views, some examples can be found in Fig. 2.5. Turnarounds find great use in animation studios, since they define a character design and enable drawing consistency among different artists. Turnarounds are often used by 3D computer graphics artists in their pre-production stage. Taking a 2D concept all the way to a finished 3D character is one of the most complex and time consuming tasks artists face to this day. Having a clear idea of the aspect of a character beforehand is fundamental, so turnarounds find great use as reference for the modeling and texturing stages.

View-Synthesis is the problem of synthesising novel views of a scene given a limited number of input views. This is a challenging problem to tackle, which can be solved by adopting state-of-the-art techniques. Recently, highly realistic results in view synthesis of a static scene have been achieved by reconstructing the volumetric information and the emitted radiance of the scene from each direction [16] [30]. Applying such view -synthesis techniques to character turnarounds may lead to the creation of a 3D representation of the drawn character which still does not exist, if not in the artist's mind. This representation could be used to render the target from any view desired, or even be converted to a 3D mesh if needed.

This novel use case may lead to interesting application scenarios in areas such as digital animation or 3D modeling. For example, instead of having the artist draw characters from many different angles, a program could handle that on its own, by having as input only a few sparse views of such character. Furthermore, since the view synthesis algorithm is able to capture the 3D structure of the drawing, a 3D mesh of the model can be obtained after a conversion process. Therefore, 3D modeling could be done completely - or in part - trough the use of hand drawn images, without the use of conventional 3D modeling software (e.g. Blender).

1.2 State of Research

View-synthesis is the problem of rendering realistic images of a scene from novel viewpoints starting out from just a limited set of images of that scene. As mentioned, rendering novel views by learning the radiance field of a scene along with its volumetric representation has proven to be an effective solution for view synthesis problems. Algorithms such as NeRF [16] and Plenoxels [30] follow this approach, and have lead to SoA results in the field.

Neural Radiance Fields (NeRF) uses a fully-connected deep neural network to represent the static scene, which can be seen as a continuous 5D function. The function takes as input a single continuous 5D coordinate, which encodes spatial location (x, y, z) and viewing direction (θ, ϕ). Its output is the volume density and view-dependent emitted radiance at that spatial location. NeRF gained a lot of attention right after its recent publication, and many papers have already been published which build upon it [1], mitigating some of its flaws, one of which is its slow convergence. In fact, NeRF takes about a day to produce SoA results.

Plenoxels represent a scene as a sparse 3D grid with spherical harmonics. This representation can be optimized from calibrated images via gradient methods and regularization without any neural components. On standard benchmark tasks Plenoxels are optimized two orders of magnitude faster than Neural Radiance Fields with no loss in visual quality, suggesting that the key element of NeRF is not the neural network but the differentiable volumetric renderer.

NeRF and Plenoxels learn a volumetric representation of the scene, or of an object if that is all that's present in a scene. Since during training the 3D structure of the object is learned, these techniques could also be converted into a mesh of such object, which could eventually be textured with the learned information on the radiance field.

To the best of the authors knowledge, character turnarounds have never been used as datasets for view synthesis tasks, nor for automatic 3D mesh reconstruction tasks. However, research which deals with drawings and modeling in general can be considered relatable. StyleProp [5] is able to give to an already existing 3D model a different style/texturing by making an artist draw over just one render of the 3D model. Although the method successfully textures the model over all its surface while keeping a high resolution, it is necessary to start from an original 3D model for the algorithm to work. Monster Mash [2] presents a whole framework for sketch-based modeling and animation of 3D organic shapes. The modeling process is interactive and can work entirely in an intuitive 2D domain, enabling a playful and casual experience. The program uses 3D inflation with a novel rigidity-preserving layered deformation model to produce a smooth 3D mesh that is immediately ready for animation. The application is an incredibly convenient tool for artists and novels, however, the algorithm is inherently limited in producing balloon-like meshes given the inflation process. Furthermore, view-synthesis algorithms could be able to model also the lighting in the character's drawing, of course only if the artists is willing to draw the character as illuminated by light in the turnaround.

1.3 Goals and Contributions

The goal of this thesis is to obtain a 3D representation of a drawn character, starting out from its turnaround. From the learned representation we want to be able to render our character from any view desired. For convenience, we also want to be able to convert such representation into a mesh. The steps taken to reach the goal, which represent the contributions, are:

- Implementing a Unity plugin for fast generation of synthetic datasets.
- Turning character turnarounds into usable datasets for view-synthesis tasks, with the help of the above plugin.
- Applying view-synthesis state-of-the-art techniques - in particular NeRF and Plenoxels - to the novel turnaround datasets created.

Before testing NeRF and Plenoxels with the turnaround dataset an assessment was performed over the viability of NeRF when few views are available and when camera positions are imprecise, in order to get an idea of what results to expect out of the turnaround dataset.

1.4 Outline

The paper will start with a theoretical background. The background will cover:

- Necessary computer graphics fundamentals.
- View synthesis and its SoA techniques, such as Neural Volumes, NeRF and Plenoxels.
- Description of what a turnaround is and what it is used for.

The author's contributions are then presented as follows:

- The followed approach.
- Description of the unity plugin implemented for synthetic datasets creation.
- Process followed for the creation of the turnarounds dataset.
- Assessing NeRF's performance over the challenging scenario imposed by this dataset.
- Results obtained from the use of turnaround datasets with NeRF and Plenoxels.
- Conclusions and ideas for future work are discussed.

Part I

Theoretical Background

2 Background

This chapter will give a general background over necessary concepts relative to the thesis. The background will mainly cover computer graphics topics ranging from classic computer graphics rendering and light transport, 3D structure representation, light fields, up to the state-of-the-art techniques in the field of novel view synthesis. Finally, a brief description of what turnarounds are is given. Much of the computer graphics background content has been taken from the amazing SoA review on neural rendering made by Tewari et. al. [24].

2.1 Classical Rendering and Light Transport

Classical computer graphics methods approximate the physical process of image formation in the real world: light sources emit photons that interact with the objects in the scene, as a function of their geometry and material properties, before being recorded by a camera. This process is known as light transport. Camera optics acquire and focus incoming light from an aperture onto a sensor or film plane inside the camera body. The sensor or film records the amount of incident light on that plane, sometimes in a nonlinear fashion. All the components of image formation—light sources, material properties, and camera sensors—are wavelength dependent. Real films and sensors often record only one to three different wavelength distributions, tuned to the sensitivity of the human visual system. All the steps of this physical image formation are modelled in computer graphics: light sources, scene geometry, material properties, light transport, optics, and sensor behavior.

The process of transforming a scene including lights, surface geometry and material into a simulated camera image is known as rendering. The two most common approaches to rendering are rasterization and raytracing: Rasterization is a feed-forward process in which geometry is transformed into the image domain, sometimes in back-to-front order known as painter's algorithm. Ray-tracing is a process in which rays are cast backwards from the image pixels into a virtual scene, and reflections and refractions are simulated by recursively casting new rays from the intersections with the geometry [27]. Hardware-accelerated rendering typically relies on rasterization, because it has good memory coherence. However, many real-world image effects such as global illumination and other forms of complex light transport, depth of field, motion blur, etc. are more easily simulated using raytracing, and recent GPUs now feature acceleration structures to enable certain uses of raytracing in real-time graphics pipelines. Although rasterization requires an explicit geometric representation, ray-tracing/raycasting can also be applied to implicit representations. In practice, implicit representations can also be converted to explicit forms for rasterization using the marching cubes algorithm [12] and other similar methods. The quality of images produced by a given rendering pipeline depends heavily on the accuracy of the different models in the pipeline. The components must account for the discrete nature of computer simulation, such as the gaps between pixel centers, using careful application of sampling and signal reconstruction theory.

In computer graphics, the simulation of light transport is a tool that helps us to create convincing images of an artificial world. Given a description of the environment - including geometry, scattering properties of the surfaces, description of the light sources, and the viewpoints from which images should be generated - light transport algorithms simulate the physics of this world, in order to generate realistic, accurate images. One of the main goals of light transport algorithms is to increase the visual quality of synthetic virtual environments. Light transport considers all possible paths of light from the emitting light sources, through a scene, and onto a camera. A well known formulation of this problem is the classical rendering equation [8]:

$$L_o(p, \omega_o, \lambda, t) = L_e(p, \omega_o, \lambda, t) + L_r(p, \omega_o, \lambda, t)$$

where L_o represents outgoing radiance from a surface as a function of location, ray direction, wavelength, and time. The term L_e represents direct surface emission, and the term L_r represents the interaction of incident light with surface reflectance:

$$L_r(p, \omega_o, \lambda, t) = \int_\Omega f_r(p, \omega_o, \lambda, t) L_i(p, \omega_o, \lambda, t)(\omega_i \cdot n) d\omega i$$

Note that this formulation omits consideration of transparent objects and any effects of subsurface or volumetric scattering. The rendering equation is an integral equation, and cannot be solved in closed form for nontrivial scenes, because the incident radiance L_i appearing on the right hand side is the same as the outgoing radiance Lo from another surface on the same ray. Therefore, a vast number of approximations have been developed. The most accurate approximations employ Monte Carlo simulations [25], sampling ray paths through a scene. Faster approximations might expand the right hand side one or two times and then truncate the recurrence, thereby simulating only a few "bounces" of light. Computer graphics artists may also simulate additional bounces by adding non-physically based light sources to the scene.

2.2 3D Shape Representations

To model objects in a scene, many different representations for scene geometry have been proposed. They can be classified into explicit and implicit representations. Explicit methods describe scenes as a collection of geometric primitives, such as triangles, point-like primitives, or higher-order parametric surfaces. Implicit representations on the other hand define a surface as the zero-crossing of a R^3 to R function, basically a level-set. However, most hardware and software renderers are tuned to work best on triangle meshes, and will convert other representations into triangles for rendering.

2.3 Interaction between Light and Scene

Modeling the interaction between light and scene is an incredibly challenging topic. The interactions of light with scene surfaces depend on the material properties of the surfaces. In computer graphics rendering pipelines, using the whole 12-dimensional light scattering function to model material properties is too complex and computationally intensive. Therefore, assumptions and simplifications are made in order to tackle the complexity of such interaction. By avoiding to model fluorescence (ability of the illuminated object to emit the incident light in the form of other electromagnetic frequencies) and phosphorescence (same thing, yet emission happens gradually), we remove two degrees of freedom. Doing this, the light scattering function has been simplified to a more tractable model representation through the bidirectional subsurface scattering reflectance distribution function (BSSRDF). By avoiding to model the subsurface scattering properties of the illuminated material we can further decrease the degrees of freedom. At this point, we are modeling light interaction with what is called the spatially varying bidirectional reflectance distribution function (SVBRDF). If we choose to treat materials as homogeneous, that is when the interaction between light and material does not change across the surface, then materials may be represented as bidirectional reflectance distribution functions (BRDFs). A BRDF is a 5-dimensional function that describes how much light of a given wavelength incident on a surface point from each incoming ray direction is reflected toward each exiting ray direction (4-dimensional if frequency is fixed). While a BRDF only models light interactions that happen at a single surface point, a BSSDRF models how light incident on one surface point is reflected at a different surface point. Again, when a BRDF changes across a surface, it is referred to as a spatially varying BRDF (SVBRDF). Spatially varying behavior across geometry may be alternatively represented by binding discrete materials to different geometric primitives, or via the use of texture mapping. A texture map defines a set of continuous values of a material parameter, such as diffuse albedo, from a 2- or 3-dimensional domain onto a surface. 3-dimensional textures represent the value throughout a bounded region of space and can be applied to either explicit or implicit geometry. 2-dimensional textures map from a 2-dimensional domain onto a parametric surface; thus, they are typically applicable only to explicit geometry.

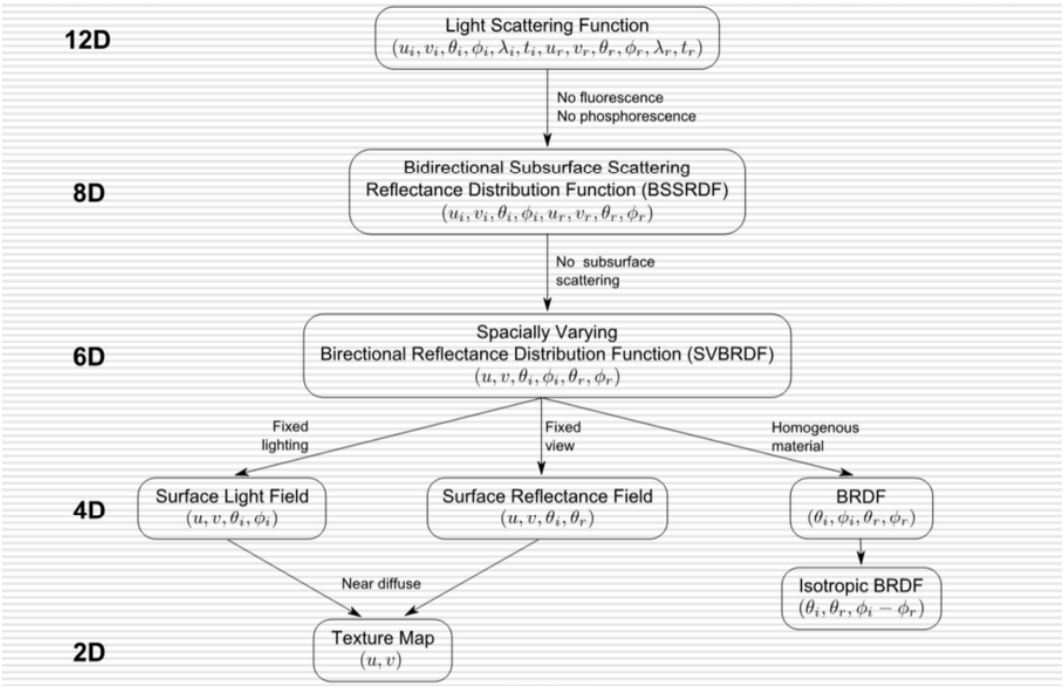


Figure 2.1: Light scattering function and its following simplifications. Out of all, BRDF is the most used material behaviour approximation for computer graphics applications.

Sources of light in a scene can be represented using parametric models; these include point or directional lights, or area sources that are represented by surfaces in the scene that emit light. Some methods account for continuously varying emission over a surface, defined by a texture map or function. Often environment maps are used to represent dense, distant scene lighting. These environment maps can be stored as non-parametric textures on a sphere or cube, or can be approximated by coefficients of a spherical harmonic basis. Any of the parameters of a scene might be modeled as varying over time, allowing both animation across successive frames, and simulations of motion blur within a single frame.

2.4 3D Shape Representations using Deep Learning

3D reconstruction can be approached using Deep Learning, in multiple ways. Discrete output representations can be used, such as voxels, point clouds or meshes. Each of these representations come with certain disadvantages due to their discrete nature. Voxels do not scale to very large resolutions, point clouds lose topology and meshes are difficult to infer of by a neural network, leading to non-water-tight reconstructions if no initial template mesh has been formed.

We can alternatively use neural networks to implicitly represent the 3D surface as the continuous decision boundary of a deep neural network classifier. This alternative representation of continuous 3D shapes as level sets can be achieved by optimizing deep networks that map 3D coordinates to signed distance functions [7] [19] or occupancy networks [13] [4]. In contrast to existing approaches, this representation encodes a description of the 3D output at infinite resolution without excessive memory footprint. This representation can efficiently encode 3D structure and can be inferred from various kinds of input.However, these models are limited by their requirement of access to ground truth 3D geometry, typically obtained from synthetic 3D shape datasets such as ShapeNet.

Subsequent work has relaxed this requirement of ground truth 3D shapes by formulating differentiable rendering functions that allow neural implicit shape representations to be optimized using only 2D images. Niemeyer et al. [18] represent surfaces as 3D occupancy fields and use a numerical method to find the surface intersection for each ray, then calculate an exact derivative using implicit differentiation. Each ray intersection location is provided as the input to a neural 3D texture field that predicts a diffuse color for that point. Sitzmann et al. [23] use a less direct neural 3D representation that simply outputs a feature vector and RGB color at each continuous 3D coordinate, and propose

a differentiable rendering function consisting of a recurrent neural network that marches along each ray to decide where the surface is located. Though these techniques can potentially represent complicated and high resolution geometry, they have so far been limited to simple shapes with low geometric complexity, resulting in oversmoothed renderings.

2.5 Light Fields

Light fields, in short, means all of the light in a space. When we say we want to capture the light field of a scene, we mean we want to capture all the light there is in it. Light fields are part of a plenoptic function, meaning that all of the light in space has five dimensions. Three dimensions are used to encode position in the 3D environment (x, y, z), while the light rays themselves have two dimensions to define where they are pointing towards (θ, ϕ). However, since light rays are infact lines in space, we actually just need four dimensions to define them, therefore light fields are 4-dimensional. Light rays are holographic in nature. This means that if we know the light field of a 3D scene, we can render what our eyes would see from a given position and direction. However, no knowledge of the 3D volume is know, just the end resulting light rays that happened to bounce on the capturing cameras.

2.6 View Synthesis and State of the Art

Novel view synthesis is the problem of generating novel camera perspectives of a scene given a fixed set of images of the same scene. Novel view synthesis methods thus deal with image and video synthesis conditioned on camera pose. Key challenges underlying novel view synthesis are inferring the scene's 3D structure given sparse observations, as well as inpainting of occluded and unseen parts of the scene.

Some approaches reconstruct a learned representation of the scene from the observations, learning it end-to-end with a differentiable renderer. This enables learning of priors on geometry, appearance and other scene properties in a learned feature space. Such neural scene representation based approaches range from prescribing little structure on the representation and the renderer [3], to proposing 3D-structured representations such as voxel grids of features [23][17], to explicit 3D disentanglement of voxels and texture [31], point clouds [14], multi-plane images [28] , or implicit functions [22] which equip the network with inductive biases on image formation and geometry. Neural rendering approaches have made significant progress in previously open challenges such as the generation of view-dependent effects or learning priors on shape and appearance from extremely sparse observations [23][28]. While neural rendering shows better results compared to classical approaches, it still has limitations, i.e. they are restricted to a specific use-case and are limited by the training data. Especially, view dependent effects such as reflections are still challenging.

View synthesis techniques - such as Neural Volumes, NeRF and Plenoxels [11][16][30] - learn the 3D structure of the static scene by the captured 2D images, along with the view-dependent emitted radiance field of such scene, so to render the scene from any view desired.

2.7 Neural Volumes

Neural Volumes [11] addresses the problem of automatically creating, rendering, and animating high-quality object models from multi-view video data (see Figure 9). The method trains a neural network to encode frames of a multi-view video sequence into a compact latent code which is decoded into a semi-transparent volume containing RGB and opacity values at each (x, y,z) location. The volume is rendered by ray-marching from the camera through the volume, accumulating color and opacity to form an output image and alpha matte. Formulating the problem in 3D rather than in screen space has several benefits: viewpoint interpolation is improved because the object must be representable as a 3D shape, and the method can be easily combined with traditional triangle mesh rendering. The method produces high-quality models despite using a low-resolution voxel grid (1283) by introducing a learned warp field that not only helps to model the motion of the scene but also reduces blocky voxel grid artifacts by deforming voxels to better match the geometry of the scene and allows the system to shift voxels to make better use of the voxel resolution available. The warp field is modeled as a spatially-weighted mixture of affine warp fields, which can naturally model piece-wise deformations. By virtue of the semi-transparent volumetric representation, the method can reconstruct challenging objects such

as moving hair, fuzzy toys, and smoke all from only 2D multi-view video with no explicit tracking required. The latent space encoding enables animation by generating new latent space trajectories or by conditioning the decoder on some information like head pose.

2.8 Neural Radiance Fields for View Synthesis

Neural Radiance Fields, or NeRF in short, presented a new method for representing complex scenes that achieves state of the art results for view synthesis. Given a set of input images of a scene NeRF optimizes a volumetric representation of the scene as a vector valued function, which is defined for any continuous 5D coordinate, consisting of a location and view direction. NeRF parameterises the scene representation as a fully connected deep network that takes each single 5D single coordinate, and outputs the corresponding volume density and view-dependent emitted RGB radiance at that location. We can then use techniques from volume rendering to composit these values along a camera array to render any pixel. This rendering is fully differentiable, so NeRF is able to optimize the scene representation by minimizing the error of rendering all camera rays from the input collection of standard RGB images.

In short, NeRF represents a static scene as a continuous 5D function using a fully-connected deep neural network. The function takes as input a single continuous 5D coordinate, which encodes spatial location (x, y, z) and viewing direction (θ, ϕ). Its output is the volume density and view-dependent emitted radiance at that spatial location.

NeRF is able to capture view-dependent effects, such as the semi-transparent appearance of an olive oil bottle, or the sharp reflections and specularities on a metal cars body and windshield. Furthermore, NeRF's representation captures high quality scene geometry and complex occlusions, such as the ones one could find in a Christmas tree leaves and ornaments, producing appealing depth renderings. This geometry is precise enough to be used for additional graphics applications, such as virtual object insertion with convincing occlusion effects.

NeRF takes as input images as well as intrinsics and extrinsics of the camera used. In real world scenarios we often don't know the intrinsics of a camera, and it is even less likely to know relative camera positions between pictures. Thankfully, this is a non-problem, since Structure-from-Motion (SfM) algorithms are able to estimate these camera parameters reliably in most scenarios. Do note that this is not always the case, especially if the camera taking the images cannot be approximated to a pinhole camera model, if images are few, if camera moves too much from one image to another or if not enough high-frequency details are present, e.g. plain color wall or clear sky. However, if working in a simulated environment, there is no need for such estimation, given that we know ground-truth camera positions and intrinsics. This enables us to better test the capabilities of NeRF, without having to worry about problems that could be given by an erroneous camera position estimation for example.

To render the learned Neural Radiance Field from a particular point the following steps are executed (in Fig. 2.2 this overall pipeline is shown):

- march camera rays through the scene, these rays will be sampled at fixed steps along the ray generating a sampled set of 3D points in the virtual space,
- use these points and their corresponding 2D viewing directions as input to the neural network [Fig. 2.3], the output will be a RGB color and density value associated by the network to that very same point and direction,
- use classic volume rendering techniques [9] to accumulate those colors and densities into a 2D image.

Because this process is naturally differentiable, we can use gradient descent to optimize this model by minimizing the error between each observed image and the corresponding views rendered from our representation. Minimizing this error across multiple views encourages the network to predict a coherent model of the scene by assigning high volume densities and accurate colors to the locations that contain the true underlying scene content.

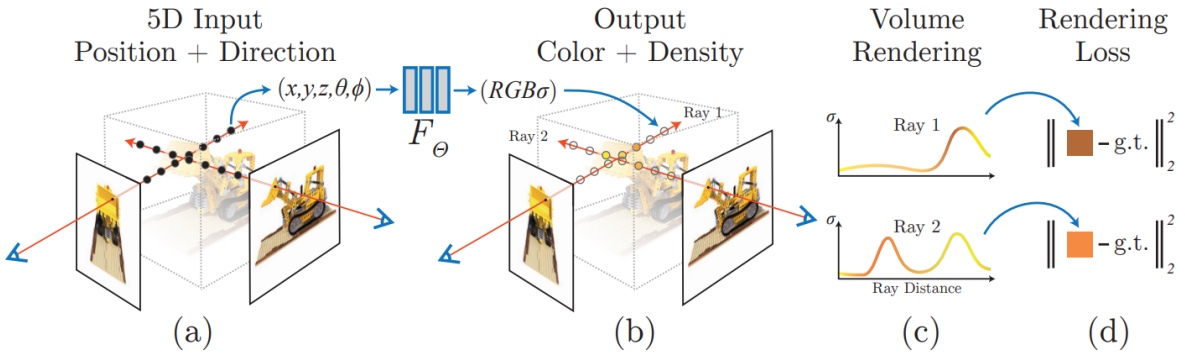


Figure 2.2: An overview of neural radiance field general pipeline, including scene representation and differentiable rendering procedure. Images are rendered by sampling 5D coordinates along camera rays (a), feeding those locations into an MLP to produce a color and volume density (b), and using volume rendering techniques to composite these values into an image (c). This rendering function is differentiable, so we can optimize our scene representation by minimizing the residual between synthesized and ground truth observed images (d).

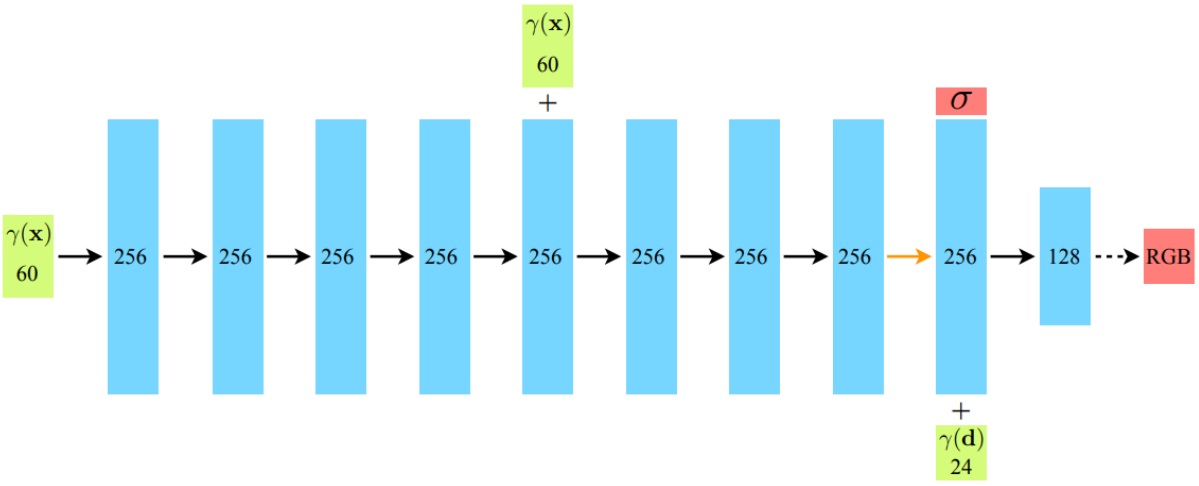


Figure 2.3: A visualization of NeRF's fully-connected network architecture. Input vectors are shown in green, intermediate hidden layers are shown in blue, output vectors are shown in red, and the number inside each block signifies the vector's dimension. All layers are standard fully-connected layers, black arrows indicate layers with ReLU activations, orange arrows indicate layers with no activation, dashed black arrows indicate layers with sigmoid activation, and "+" denotes vector concatenation. The positional encoding of the input location ($\gamma(x)$) is passed through 8 fully-connected ReLU layers, each with 256 channels. The DeepSDF [20] architecture is followed by including a skip connection that concatenates this input to the fifth layer's activation. An additional layer outputs the volume density σ (which is rectified using a ReLU to ensure that the output volume density is non-negative) and a 256-dimensional feature vector. This feature vector is concatenated with the positional encoding of the input viewing direction ($\gamma(d)$), and is processed by an additional fully-connected ReLU layer with 128 channels. A final layer (with a sigmoid activation) outputs the emitted RGB radiance at position x, as viewed by a ray with direction d.

To render the color of any ray passing through the scene NeRF uses principles from classical volume rendering [9]. The volume density $\sigma(x)$ can be interpreted as the differential probability of a ray terminating at an infinitesimal particle at location x. The expected color $C(\mathbf{r})$ of a camera ray $\mathbf{r}(t) = o + t\mathbf{d}$ is approximated by integrating over samples along the ray:

$$\widehat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - exp(-\sigma_i\delta_i))\mathbf{c}_i$$

$$where \quad T_i = exp\left(-\sum_{j=0}^{i-1}\sigma_j\delta_i\right)$$

T_i represents how much light is transmitted through ray $\mathbf{r}$ to sample i (versus contributed by preceding samples), $(1 - exp(-\sigma_i\delta_i)$ denotes how much light is contributed by sample i, σ_i denotes the opacity of sample i, and $\mathbf{c}_i$ denotes the color of sample i, with distance δ_i to the next sample. Although this formula is not exact (it assumes single scattering and constant values between samples), it is differentiable and enables updating the 3D model based on the error of each training ray.

A confrontation with other SoA view synthesis techniques on synthetically rendered data follows. Scene representation networks [23], SRN in short, implicitly represent the scene using a fully connected neural network. However, SRN have an issue with multi-view consistency, and are not able to represent high frequency level details. Local light field fusion [15] uses a pre-trained network to promote each input view to a high resolution 3D volume. Inconsistencies between adjacent volumes causes flickering artifacts. Neural Volumes [11] encode a scene as a 128 cubed voxel grid, and use a work field to better allocate these limited samples. However, they are still unable to represent high resolution content. NeRF qualitatively and quantitatively outperforms the other methods.

2.9 Plenoxels: Radiance Fields without Neural Networks

Neural networks are not the only way to learn the 3D volume and the radiance fields behaviour of bounded or unbounded scenes. In particular, faster training and inference has been achieved by Plenoxels [30]. A Plenoxel is a plenoctic volume element that represents neural radiance fields without using neural networks. Plenoxel uses an explicit volumetric representation, based on a view-dependent sparse voxel grid, avoiding the need for a neural network. This model can render photorealistic novel viewpoints and be optimized end-to-end from calibrated 2D photographs, using the differentiable rendering loss on training views as well as a total variation regularizer. Because there are no neural networks, plenoxels achieve similar results as NeRF while optimizing much faster then NeRF. Just after a few seconds, plenoxels captures most of the geometry and appearance. In general, Plenoxels achieve NeRF quality with a speedup of two orders of magnitude (i.e. what takes ten minutes for plenoxels takes a day for NeRF).

Like NeRF, the plenoxel model is optimized using calibrated images of a scene. A graphical representation is presented in Fig. 2.4 and detailed hereafter. To render a pixel, a ray is projected from the pixel towards the model. The ray intersects with the voxels in the model. Each voxel stores opacity and spherical harmonics coefficients. The plenoctic function is evaluated at regular intervals along the ray using trilinear interpolation of the neighboring voxel coefficients. Finally, the ray color is determined by combining these sample values according to volume rendering formula. We optimize our plenoxel model using gradients computed on mini batches of training rays. Finally, empty voxels are pruned away to produce a sparse representation and sub divide non empty voxels to achieve high resolution.

For volume rendering, the same differentiable model found in NeRF is used, where the color of the ray is approximated by integrating over samples along the ray. Results obtained with plenoxel show that the key component in NeRF is the differential volumetric rendering, not the neural network. In addition it is found that trilinear interpolation is key to achieve high resolution and better convergence. Regularizing space is also important, particularly when the number of observations is low.

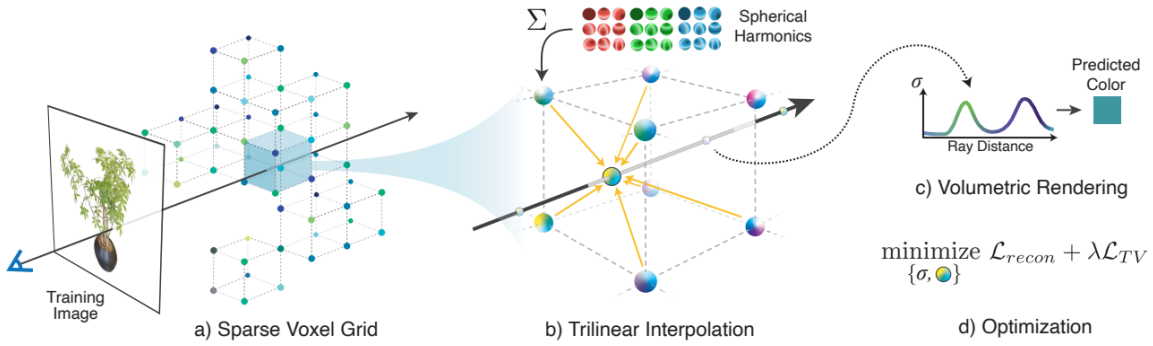


Figure 2.4: Overview of the sparse Plenoxel model. Given a set of images of an object or scene, a sparse voxel grid, or "Plenoxel", is reconstructed (a) with density and spherical harmonic coefficients at each voxel. To render a ray, it computes the color and opacity of each sample point via trilinear interpolation of the neighboring voxel coefficients (b). The color and opacity of these samples is integrated using differentiable volume rendering (c), just as NeRF does. The voxel coefficients can then be optimized using the standard MSE reconstruction loss relative to the training images (d), along with a total variation regularizer.

2.10 Turnarounds and Character Model Sheets

Character turnarounds are a way for an artist or animator to gain a feel for the character that they are designing by sketching out different positions and views of the body. This is to create a point of reference during production to know the right body shapes and looks when drawing a character in different poses and angles. Therefore, turnarounds are often used in animation studios for defining a character design and to keep drawing consistency among different artists. The number of views in which a character is drawn depends on the application and the importance of the project, in general 5 views are considered sufficient for most purposes.

Model sheets are very similar to turnarounds and have the same structure. Model sheets are more used for character motion and design rather than size and proportions. These allow artists to communicate the way that characters move and how the body would flow if the character would perform a certain action, or how the character looks in different poses.

In figure 2.5 some turnarounds and model sheets have been gathered, to get the general idea.

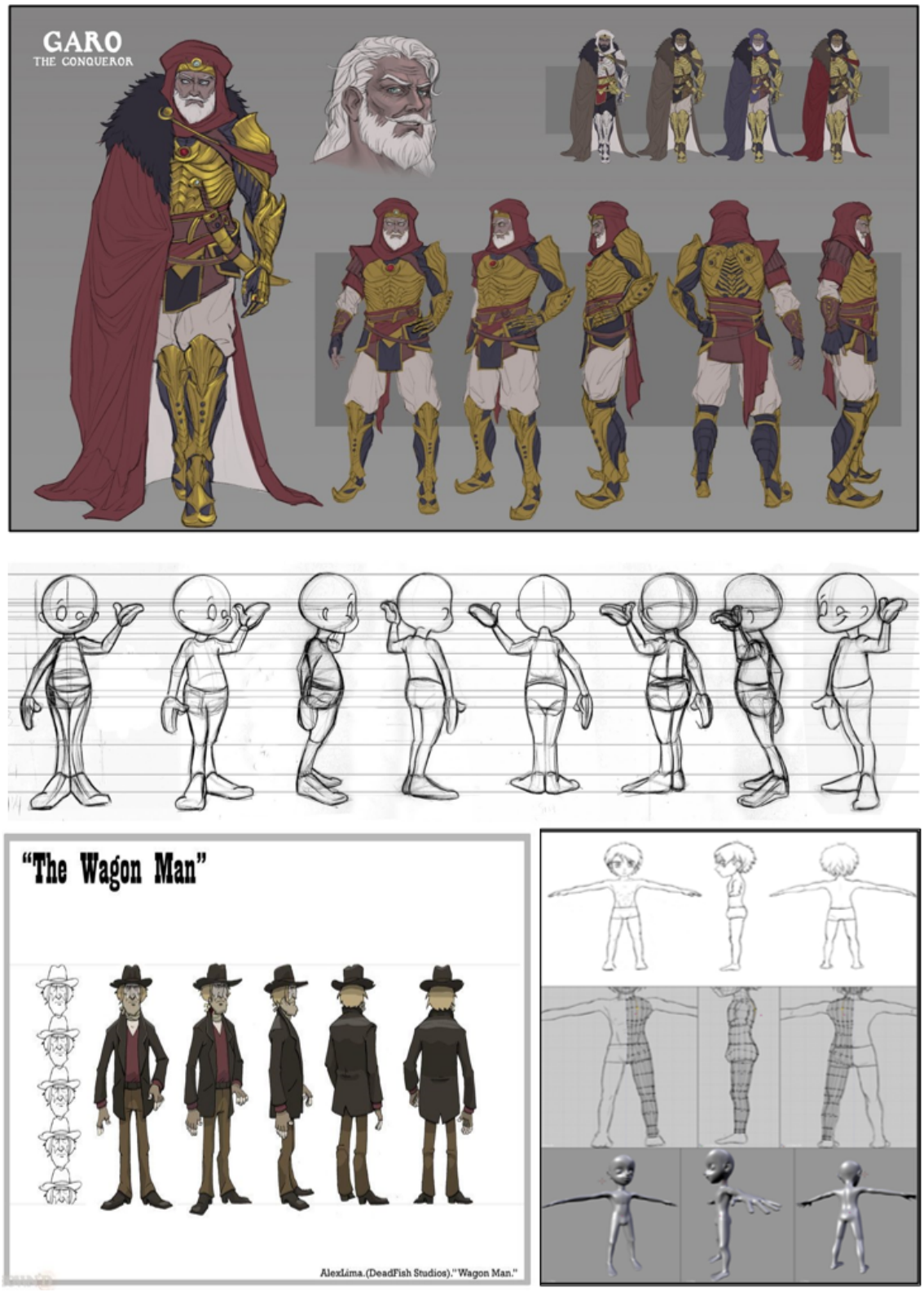


Figure 2.5: Some examples of character turnarounds and model sheets. On the lower-right, a turnaround is used as reference for 3D modeling.

Part II

Contributions

3 Approach

Here I explain the followed approach for the novel application of view synthesis on hand drawn characters, more specifically, on their relative turnaround sheet. The steps followed are here detailed:

- A Unity plugin for fast and easy synthetic datasets for NeRF and Plenoxels [16][30] is created. Such plugin is presented and explained in detail in Chapter 4. This plugin will reveals itself useful also for the creation of turnaround datasets in the next step.
- Turnarounds are divided into single images and supplied with the needed camera parameters. The process to transform a turnaround into a viable dataset for novel view synthesis will be thoroughly described in Chapter 5. In order to supply the single turnaround views with good guesses of intrinsics and extrinsics camera parameters, the author uses the unity plugin previously described.
- Since using few images and using imprecise camera positions puts NeRF and Plenoxels into a challenging scenario, a brief assessment on how they might behave when these conditions are met is done in Chapter 6.
- The novel dataset will be tested with SoA techniques in view synthesis, NeRF and Plenoxels. A general understanding can be grasped from Fig. 3.1. Details on the experimental setup, as well as results and considerations over the latter will be discussed in Chapter 7. The objective is to obtain a final 3D representation of our depicted character, capable of being rendered from multiple views with a decent level of qualitative quality. Renders will be performed from many different angles, showing the limitations of such algorithms if any.
- The obtained 3D structure of our character will not be in the form of a mesh, as already mentioned while discussing on NeRF and Plenoxels in Chapter 2. So, as a final step, such mesh will be obtained from the resulting 3D structure inferred by NeRF with the use of the marching cubes algorithm.

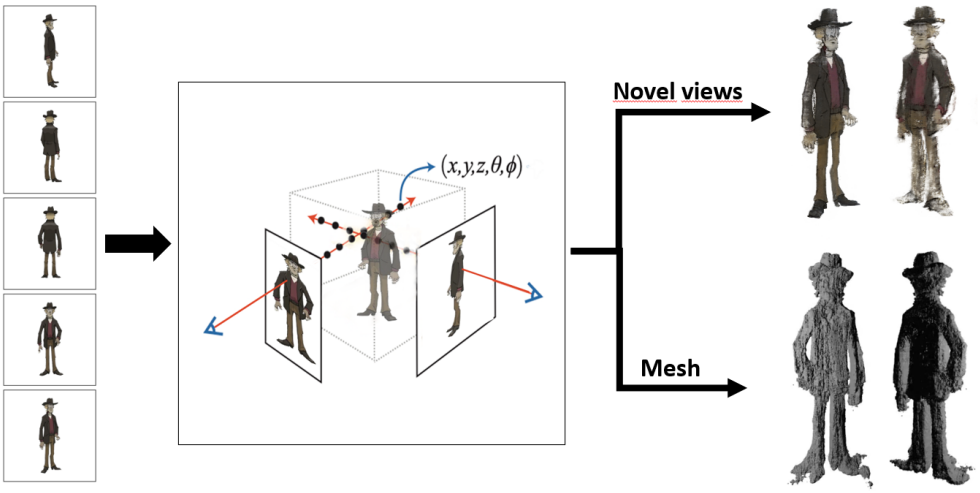


Figure 3.1: Scheme of the general framework. On the left, images taken from a character turnaround. These images will be used in the training process of NeRF and Plenoxels, in order to infer the 3D volume and light field of such character. This 3D representation of our scene will be used to render our character as seen from novel views or to infer its mesh.

4 Synthetic Dataset Generation for View Synthesis using Unity3D

A Unity plugin for easy creation of synthetic datasets for view synthesis is implemented. The plugin is not indispensable for creating turnaround datasets, however it offers many advantages over creating such datasets "by hand". The dataset can be downloaded from: `https://github.com/AndreaMas/nerf-dataset-creator-plugin`

The plugin spawns cameras around a target in the Unity scene, capturing images from them. A simple GUI is provided [Fig. 4.1] to help the user choose: which game object to target; how many cameras are spawned and in what configuration; how many images are wanted for training testing and validation. The output file contains the captured images, the ground-truth position and orientation of such cameras, depth and normal maps for final results evaluation purposes. The output dataset file is identical to the Blender data-sets used in the original NeRF paper [16], which many other view-synthesis papers share for better results comparisons.

The plugin gives to the user visible feedback to where the cameras are positioned and high customization capabilities on the positioning and intrinsics of such cameras. For these reasons, its use is highly suggested to create turnaround datasets. Furthermore, it allows to have reference images so to better evaluate if the guessed turnaround's field of view and views positions might work or not.

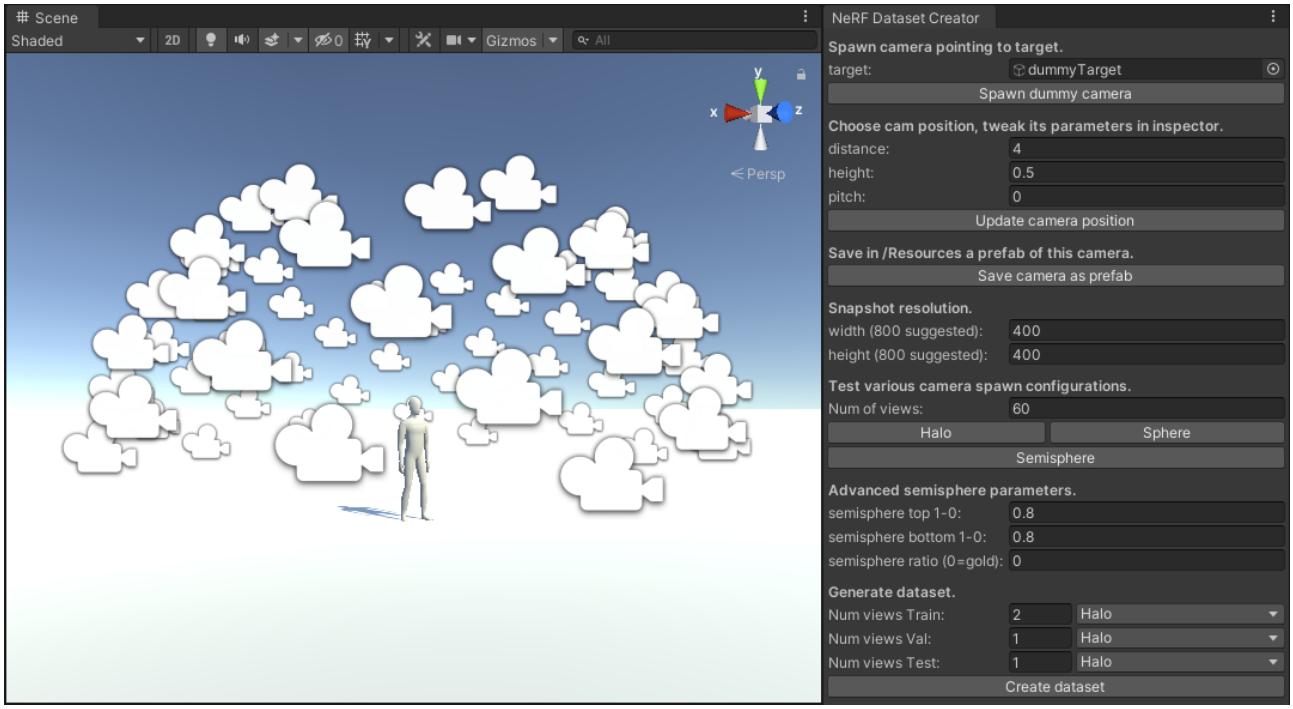


Figure 4.1: The plugin's editor window in Unity. The GUI is divided visually in 6 sections, each with a brief description of what its purpose is and with a few parameters and buttons. The User can follow and execute the descriptions in bold, from top to bottom. If any further clarification on the use is needed, the user can refer to section 4.3.

4.1 NeRF's and Plenoxels' Synthetic Dataset Structure

Before creating our own turnaround datasets, lets understand of what a standard dataset for NeRF and Plenoxels is made of. As mentioned, both take as input images as well as intrinsics and extrinsics of the camera used. However, for now, NeRF and Plenoxels (and most of the presented view synthesis SoA papers) only work when it is possible to approximate the camera model to a pinhole camera model. Over my thesis project, I've followed the original "Blender" dataset structure, used in both NeRF and Plenoxels paper for bounded synthetic scenes. However, the user is free to implement its own dataset structure, as long as the above information is then processed to be given as input to the algorithms.

The Blender dataset structure stores images for training, testing and validation in separate files [Fig. 4.2]. Furthermore, it stores camera intrinsics and extrinsics in a json file, which contains the number of views acquired of the target, the field-of-view of the camera, and for each image the camera position in the virtual space. The camera position in the json file is represented by a camera-to-world matrix. This matrix embeds within itself the information to roto-translate the camera from a default camera position to its actual position in the virtual world [Fig. 4.3]. In computer graphics this default camera position and orientation may vary. The standard followed by OpenGL, Unity and other is to have as the default camera position a camera placed in the origin, facing toward the negative z axis, with the positive x axis at its left and the positive y axis on top.

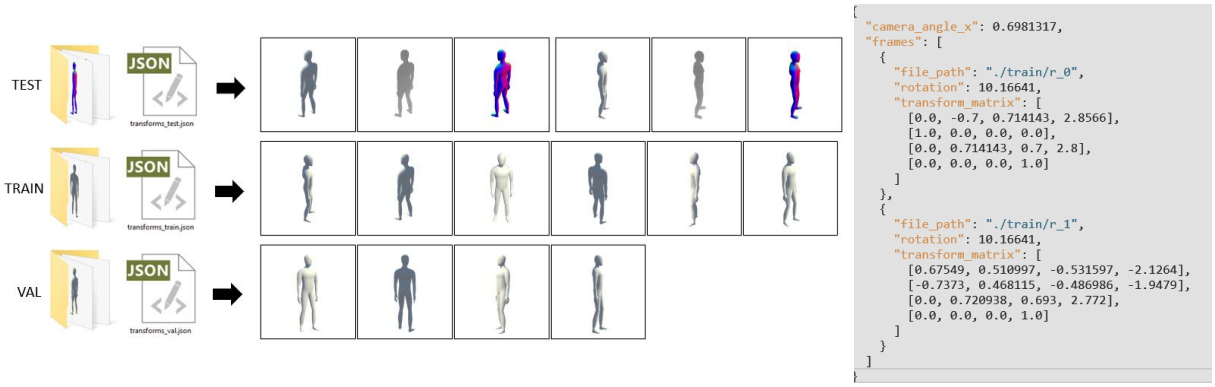


Figure 4.2: An example of Blender dataset structure. On the left, three folders containing the respective train, test and validation images. Test set has depth and normals only for results evaluation purposes. A json file defines for each set the field-of-view and extrinsics of cameras. On the right, an example of json file, which reports that the train set is made by two images, r_0 and r_1, the synthetic camera's field of view (the "camera angle x" variable), and the transform matrix, which tells us the camera position and orientation in space.

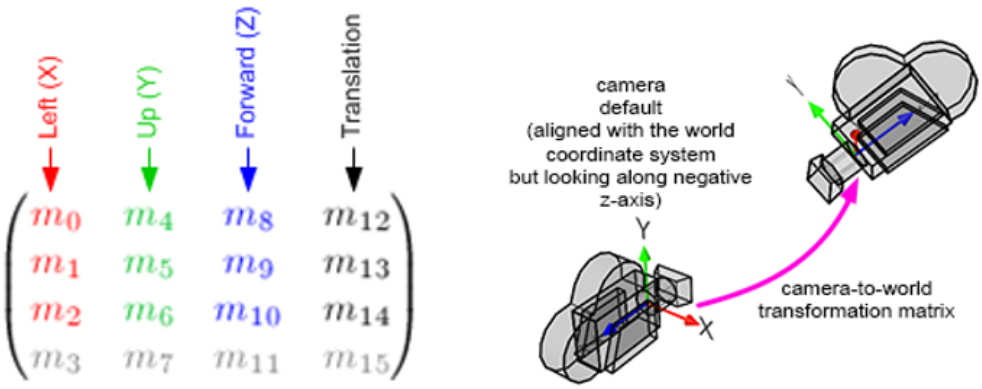


Figure 4.3: On the left, a camera-to-world matrix. On the right, its effect when applied on a camera centered at the origin of the global coordinate system and aligned with its axis. The matrix represents the roto-translation applied to the camera object.

What happens when we apply a camera-to-world matrix to the camera default position? The camera starts at the origin of the world space, with orientation as in figure. The black values in the "Translation" column represent the new coordinates of the camera. In this case, the camera is positioned in $(0, 0, 0)$ and is translated to (m_12, m_13, m_14). The red values in the "Left(x)" column represent where the x unit vector ends up after the transformation. In this case, the x unit vector starts as $(1, 0, 0)$ and becomes (m_0, m_1, m_2). Likewise is true for other axes. Last row remains $(0, 0, 0, 1)$ for roto-translations, and would change only for non linear transformations.

4.2 How the Plugin Works

The plugin consists in a couple of scripts that spawn cameras around a target, giving as output the images captured by the cameras, as well as files containing information about their intrinsic and extrinsic parameters. This is repeated three times to create train, val and test sets. Do note that the following steps are a possible framework to build NeRF/Plenoxels synthetic datasets, not strict guidelines. In fact, instead of creating multiple virtual cameras, the same camera could be used and moved around the target. The author opted for more cameras for better visualization. The plugin can be found in the project's GitHub page.

More specifically, the script works as follows:

1. first, the user is required to make the following choices:
   - user chooses a target, the desired 3D model to capture,
   - script spawns dummy camera to let the user customize its intrinsics (e.g. FoV),
   - user chooses the distance from where to take the images of the target,
   - user chooses number of images desired (for train, val and test sets).
   - user chooses in what formation to spawn cameras around target: halo, sphere or semi-sphere (for train, val and test sets)
2. a "Dataset" folder is created, as well as "Train", "Val" and "Test" sub-folders within it. These will be used to store images belonging to each different set [Fig. 4.2],
3. the following operations are repeated for each set (train, val and test):
   - cameras spawn around the target, with the chosen intrinsics, at the given distance and in the chosen formation.
   - intrinsic and extrinsic parameters of each camera are saved in a json file,
   - each camera captures an image (note: only during the image acquisition for the test set, also depth and normal images are captured, these will not be used by NeRF but might be useful to the user to better estimate NeRF's view synthesis quality),
4. the program ends, the user can move and store the Dataset folder where he pleases.

4.3 Guidelines over the Plugin's use

If the reader chooses to use the presented Unity plugin, the following guidelines and insights might be useful:

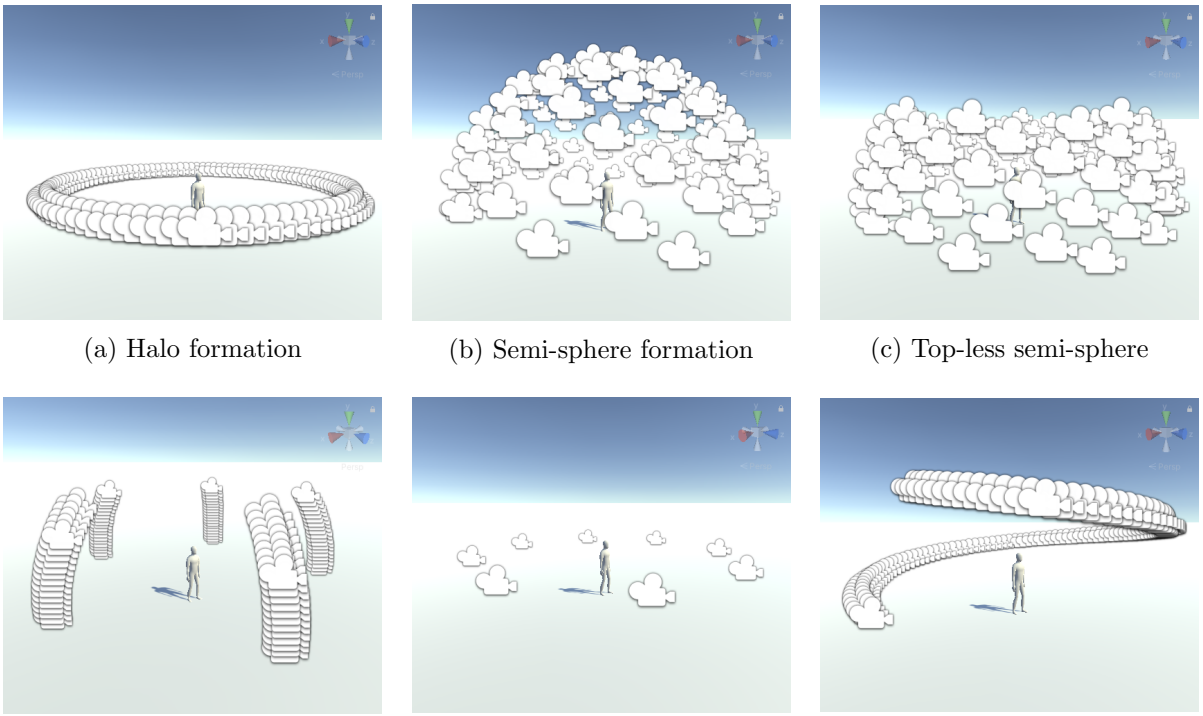
- The plugin can be integrated into Unity by drag-and-drop inside the Unity's Assets folder.
- To help the user, a GUI has been implemented, in the form of an Editor window [Fig. 4.1]. The window can be opened from Unity's Window tab (top left).
- Before the plugin can be used, it is necessary to create a new Unity Tag called "ProCam". To do so, go to Edit > Project Settings > Tags and Layers > Tags > + and write ProCam.
- The target gameobject can be assigned to the target variable trough drag-and-drop. When cameras will spawn, these will point to this gameobject.

- The user is free to assign as target an empty gameobject, using it as a dummy. This might be useful if the user wants cameras to point not exactly to the object's pivot point. Cameras will point to the dummy's location instead, which can be placed wherever the region of interest is.
- The button "spawn dummy camera" spawns a camera which the user can customize to choose the right intrinsics (such as the correct field of view) and in general to better fit its needs. In fact, if desired, additional components can be added to the camera. By clicking "save camera as prefab" the camera is saved as a prefab. It's this camera that will be spawned around the target and used to capture images of it.
- If NeRF is trained using "blender" as "dataset-type", then the images given as input are required to only have the target rendered, not the background (i.e. background must be transparent). The plugin automatically captures a transparent background (whatever the Clear Flags value in the Camera component). However, be sure to capture only the desired target (by disabling unwanted components, or by playing with the Culling Mask of the camera component).
- If any bug or error shows up, most of the times deleting the camera prefab stored in "Resources" solves the problem.
- Users can choose the resolution of the images acquired. Although any resolution is fine, working with 800x800 images is preferable when testing out NeRF for the first time. Alternatively, use 400x400 or 1600x1600 if the default resolution is respectively too low or high.
- Sphere and semi-sphere formations place cameras around the target using the golden ratio sphere sampling algorithm.
- The advanced semi-sphere parameters should be changed only if the existing camera formations around the target are not satisfactory [Fig. 4.4]. By tweaking these parameters, the user can for example place cameras only on a subsection of the semi-sphere, avoiding to capture the target from the top. Although changing these parameters produces non-intuitive changes in the camera formation, many different configurations can be generated by playing with these. Do note that setting the variable semi-sphere ratio to 0 actually sets it to the *golden ratio* constant.
- Once the "create dataset" process is done, the Dataset can be found in the Assets folder. Suggestion is to move such file out of the Assets folder quickly enough, otherwise Unity will start creating the relative meta files for each image, stalling the engine for a good amount of time and filling the dataset of unwanted files.

Unfortunately, once all images are captured and the Dataset is successfully created, if the user does not move them out of the Assets folder quickly enough, Unity will start creating the relative meta files for each image. The author has not found any way to avoid this unnecessary process, which stalls the engine for a good amount of time (depending on the amount of images taken) and fills the dataset of unwanted files.

4.4 Testing datasets generated by the plugin with NeRF

In Fig. 4.5 I present a custom synthetic dataset made with this plugin, while in Fig. 4.6 the results obtained by NeRF are shown. Results have been obtained by running the original tensorflow implementation of NeRF on an Nvidia RTX 2070 Ti. The dataset has 100 images for the training set, 15 for the validation set and 3 for the test set. The training was stopped at iteration 100000, with parameters $Nsamples = 64$ and $Nimportance = 128$. The training lasted about 6 hours.



(a) Halo formation (b) Semi-sphere formation (c) Top-less semi-sphere

(d) Column stacked (e) Eight cameras (f) Spiral

Figure 4.4: Different possible camera spawning configurations. All cameras are equally far from target and look towards it. Except for halo formation, camera positions are chosen following the golden ratio sphere sampling algorithm. Such different configurations are created modifying this algorithms parameters through the GUI in the "advanced semiphere parameters" section.

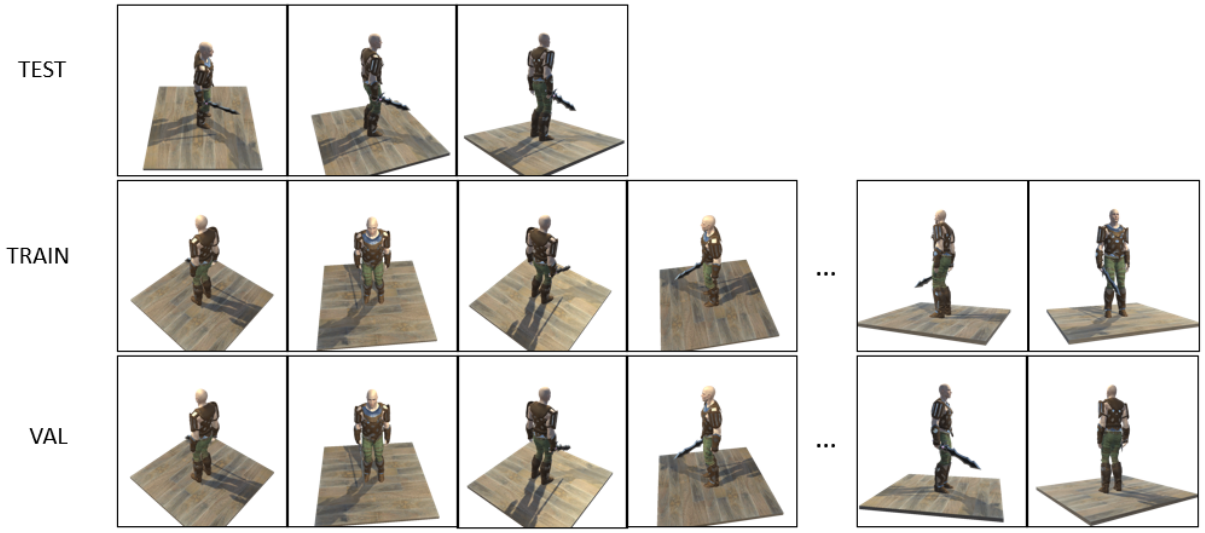


Figure 4.5: A custom synthetic dataset made with the presented plugin. The dataset has 100 images for the training set, 15 for the validation set and 3 for the test set.

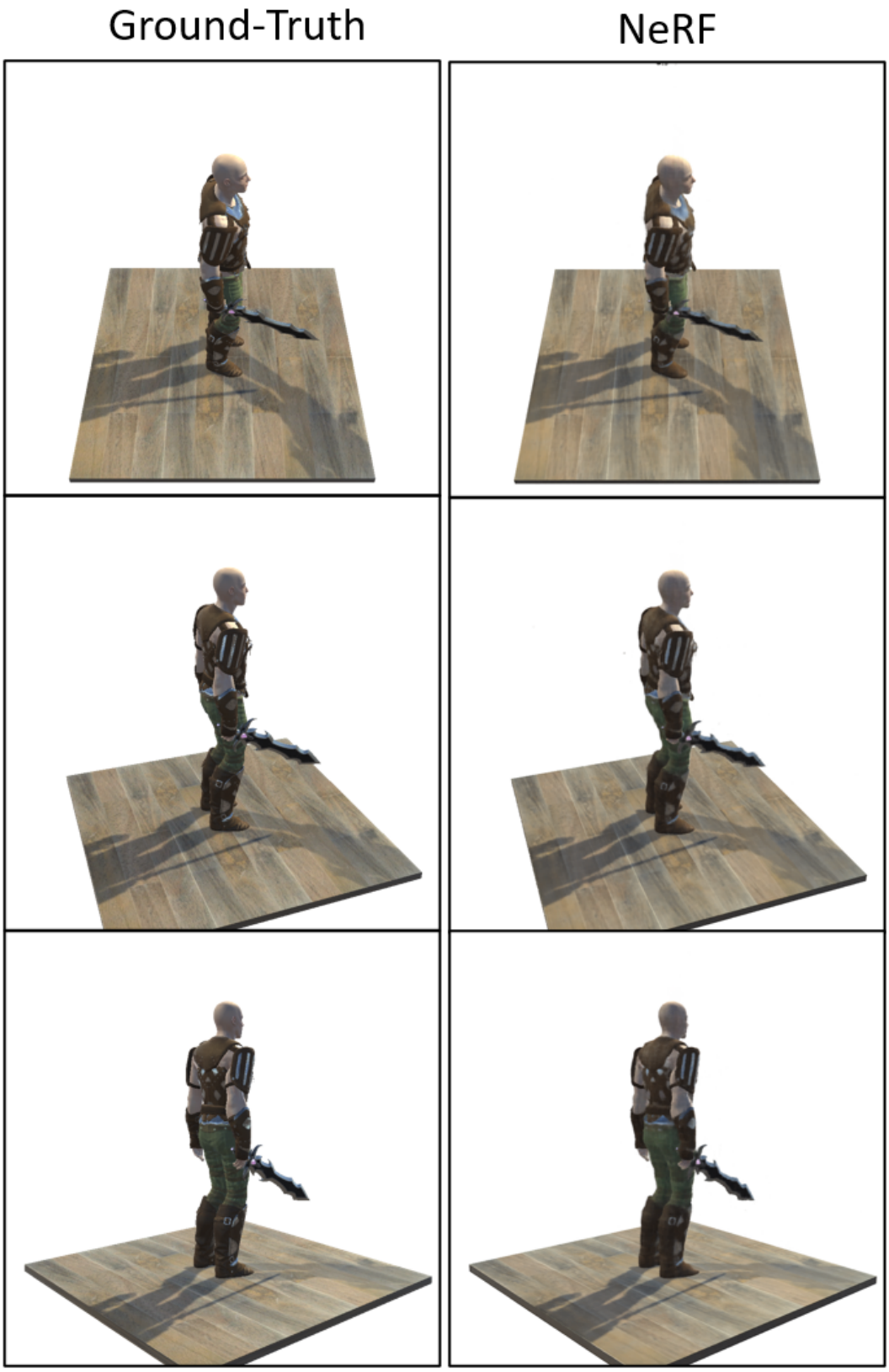


Figure 4.6: Screenshot of NeRF's result over the test set. We can see that NeRF reconstructs successfully the scene seen from views on which it was not trained on. Therefore NeRF did indeed understand the 3D structure of the scene and its view dependent lighting.

5 Novel Turnaround Datasets

The creation of the turnaround datasets is the key contribution of my thesis. The idea and hope is that given this dataset - composed by the artists drawings of the same static subject from multiple views - as input to the previously detailed SoA novel view synthesis algorithms [16][30] these will be able to render the subject from any desired view.

In the following, we will see what characteristics a turnaround needs in order to work as a dataset, what a dataset for NeRF and Plenoxels looks like, finally the process to create it.

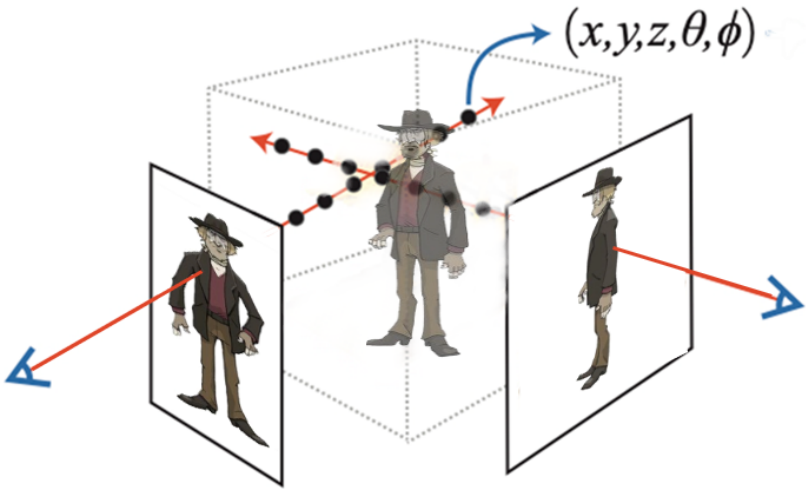


Figure 5.1: General idea of how NeRF - or similar view-synthesis algorithms - should infer a 3D representation of our character out of the different views in our turnaround.

5.1 What a viable Turnaround looks like

Some considerations on the viability of a particular turnaround as a dataset need to be made so to better get an understanding on what to expect from the novel view reconstruction:

- the more views the better. Five to eight views can be enough, but since the algorithms will reconstruct the structure and colors by enforcing consistencies among the different views, having many consistent and sparse views is crucial for the end result's quality.

- since no prior about the nature of the subject is given (nor that the subject is a drawing), it is obvious that the reconstruction of regions which are not depicted by the turnarounds will fail in the reconstruction process. These regions will also suffer of poor color consistency and quality.

- Structural and color complexity of the character is not a problem. However, the number of views needed to infer such complexities might be higher.

- In general, turnarounds depict a subject as if it was in a 3D space. This is a good thing, since we want to infer such 3D representation that was in the artists mind while drawing the character. However, this is not always the case, and some turnarounds are created depicting the character as a 2D entity, as if a character was depicted using an orthographic camera. In these cases, the reconstruction will likely give unsatisfactory results.
- Since the final dataset must have transparent background (infact, images will need to be in PNG format) its best if turnarounds have transparent or easily removable (e.g. plain color) backgrounds.

The author has chosen, after the above considerations, to work with the three turnarounds in Figure 5.2 These turnarounds where then turned into datasets for NeRF and Plenoxels. The process to do this is described in the following section 5.3.

5.2 Challenging aspects of the Turnaround Datasets

Before creating our own turnaround datasets, its necessary to know what a standard dataset for NeRF and Plenoxels is made of. This has been already described in section 4.1. As mentioned, both NeRF and Plenoxels take as input images as well as intrinsics and extrinsics of the camera used, following the original "Blender" dataset structure. Images for training, testing and validation are stored in separate files [Fig. 4.2]. Json files (one for each set) store the number of views acquired of the target, the field-of-view of the camera, and for each image the camera position in space. The camera positions in the json file are represented by camera-to-world matrices. This matrix embeds within itself the information to roto-translate the camera from a default camera position to its actual position in the virtual world 4.3. In computer graphics this default camera position and orientation may vary. The standard followed by OpenGL, Unity and other is to have as the default camera position a camera placed in the origin, facing toward the negative z axis, with the positive x axis at its left and the positive y axis on top.

Since often turnarounds are made up of few images, all of these are used in the training dataset. In fact, no validation and testing datasets are created. Just to avoid error messages from the original NeRF tensorflow implementation, one can create test and val sets equal to the training one. This means that we will have no quantitative measure to let us know if we are overfitting or not.

In view-synthesis overfitting can be seen as rendering really well our subjects appearance, but only from the views on which it has been trained on. Therefore, when rendering from novel views, the quality of the render should be much worse than expected. In general, it is very important to use validation and test sets to avoid overfitting, since one would not know in what instances the predictions made by the machine learning algorithm may fail. However, in our particular case, we are able to render a video around our target moving our camera 360 degrees around the subject. This render can be repeated at fixed intervals over the whole training, so to get a good qualitative idea of whether the training is working or not, and if we are overfitting or not.

An assessment over NeRF's and Plenoxels' behaviour when few images with imprecise camera positions are given is done in Chapter 6.

5.3 Process to Create Turnaround Datasets

The turnaround dataset will need to give to our view synthesis algorithms the images of the drawn character, so to reconstruct its 3D appearance.

Since the images in the dataset will need to have a transparent background, the turnaround must be converted into PNG format (if not already in this format), and any background will need to be substituted with transparency. At this point we create our turnaround dataset by cropping the turnaround views into individual captures. For ease of use its suggested to use quadratic captures, such as 800x800 or 1600x1600. It is crucial that the depicted character is centered in the middle of the image, as if a camera was turning around the character. Therefore, careful cropping is suggested, as good results will heavily depend on this step.

Once images are stored in the training folder, we need to tell the algorithms from where these images where "captured". In other words, we need to supply the algorithm with the position from

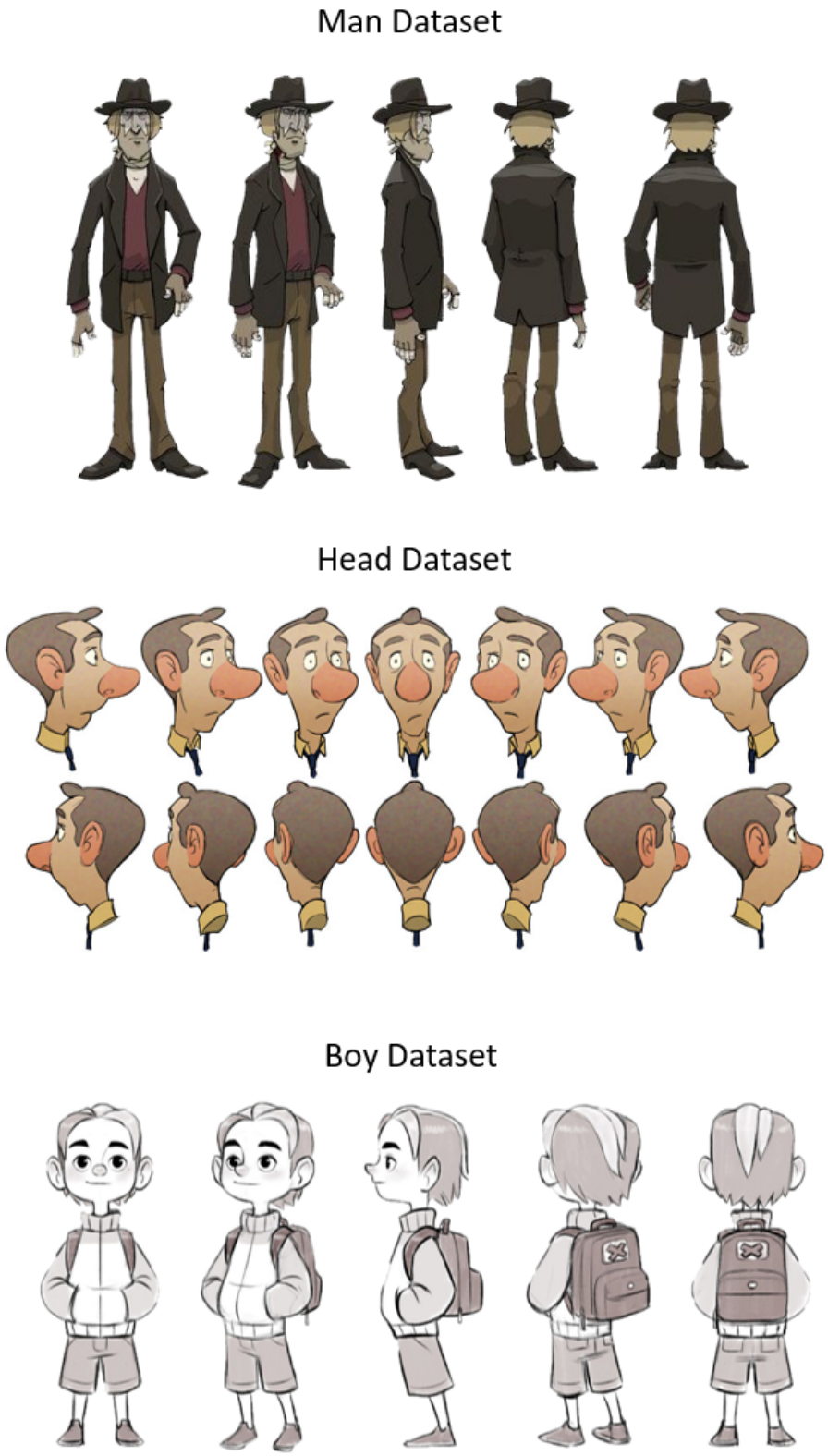


Figure 5.2: These three are the turnarounds the author has chosen to work with. These are turned into three different datasets, respectively called "Man", "Head" and "Boy" datasets.

which the character has been depicted. Unfortunately we do not have that information at our disposal. However, some characteristics of the turnarounds come in handy in this step. In general, characters in turnarounds are drawn as seen from specific viewpoints. Drawings of a character mostly capture it as seen from the front, the back, from the left and right, and finally in the in-betweens of such views. So, the character is drawn from eight different views around the target. This means that we have an approximate idea of the imaginary camera positions that are viewing the character. This of course is not by any means a precise position, but is a good initial guess.

Unfortunately, we don't know the distance from the target either, nor the field-of-view of the imagined camera. These may be guessed from the drawings by eyeballing the amount of perspective distortion in the image. In general, in turnarounds, characters are drawn with little (if any) perspective distortion, so guessing our imaginary camera capturing the scene from a good amount of distance and with a relatively narrow field-of-view is best. Still, guessing the right values of position, distance and field of view can be quite hard, and often will bring to non-satisfactory results.

The best option would be to let an algorithm estimate such values, using for example Structure-from-Motion algorithms, which happen to do just that. However, SfM algorithms are able to do this only when a good amount of images capturing the same scene using the same camera is available. Having few views of our character and having to deal with small inconsistencies from one view to the other of the turnaround, Structure-from-Motion algorithms failed to guess the position, orientation and field-of-view of our imaginary cameras.

To tackle this issue, the author has chosen to use the previously presented Unity Plugin Tool to help the user make a decent guess of possibly good camera parameters, while automating most of the steps for the dataset creation process at the same time.

First, a manikin of any kind can be imported (or created) into Unity. This dummy can be used as a reference to test and play with the cameras positioning and field-of-view. Since Unity is a game engine, the user has immediate real time feedback on what the actual parameters look like on the dummy. After some test and trial, the user can come up with a decent guess on what field-of-view and distance from target to choose [Fig 5.3, 5.4].

Furthermore, once we are satisfied with the chosen parameters, we can use the previously presented plugin scripts to automate the json file creation responsible for storing such parameters, which will then be used by the view-synthesis algorithms.

Once the dataset capturing the dummy has been created, all that is up to the user is to substitute the training images of the dummy with the turnaround ones. If extra views of the dummy have been captured, these views can be removed. It's best to also remove the relative camera parameter information in the produced json file, since this will be used by the view-synthesis algorithms to check which images to import. In the end, we will have something like the dataset in Fig 5.5.

5.4 Avoiding Possible Overfitting Artifacts

As we mentioned in section 5.2, given how few images we are using for training, we are vulnerable to overfitting. In view-synthesis (at least for view-synthesis algorithms which try to solve the problem by passing through a 3D representation) overfitting can be seen as being able to reconstruct perfectly the scene when viewed from positions present in the training set, yet failing to reconstruct the scene as viewed from other positions. This is true also for NeRF and Plenoxels [16][30]. One way this can happen is by inferring the presence of a character (or parts of it) right in front of each camera or view. This means that the character will be correctly rendered from the view in training, but a really small variation in position will show the artifact. To counter this problem, the dataset can be expanded by using the same training image more than once, but with slightly modified extrinsics. Since the appearance of our character will not change significantly with slight movements of the camera, we can use this to our advantage to counter the aforementioned issue. So, for example, starting from 5 training images, we could end up having 25, by using the same images but with five different - slightly changed - camera positions.

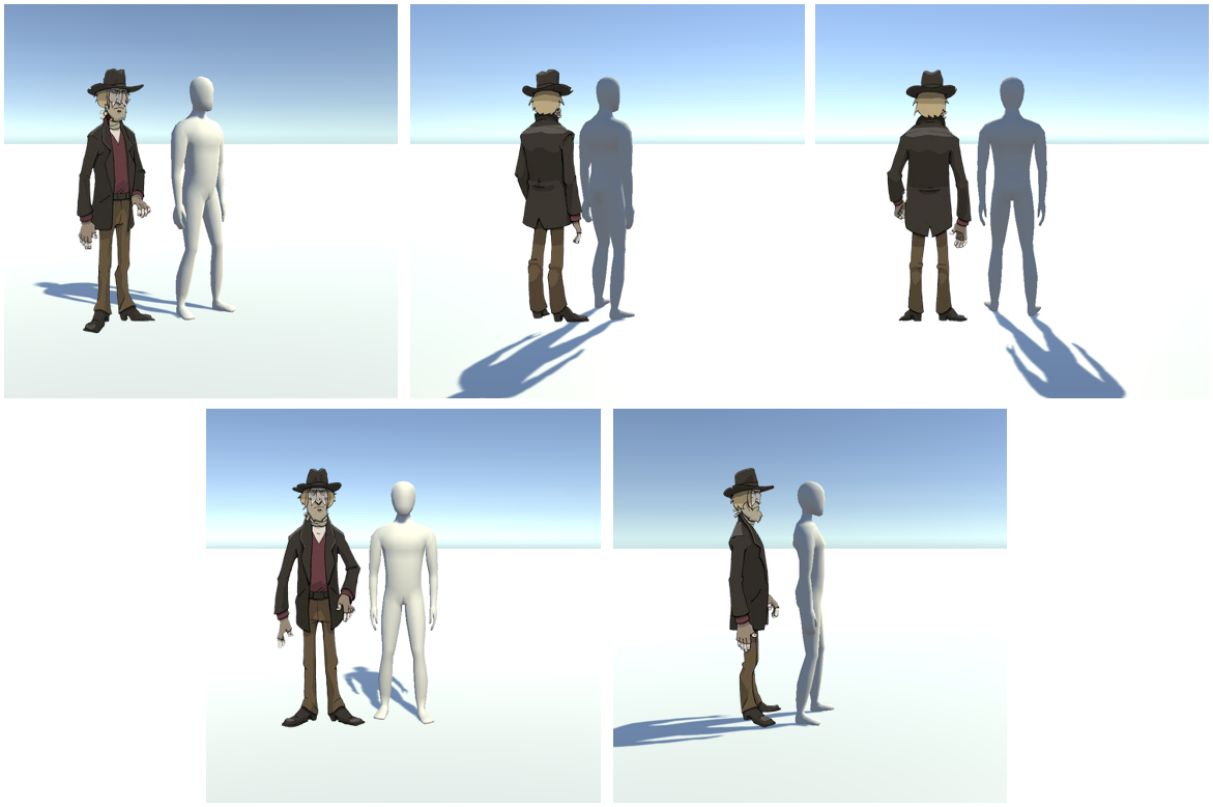


Figure 5.3: Images of overlapping turnaround and manikin images. The 3D model of the manikin has been imported into Unity and captured from different views. We can see that, after some test and trial, the user can come up with a decent guess on what field-of-view and distance from target to choose.

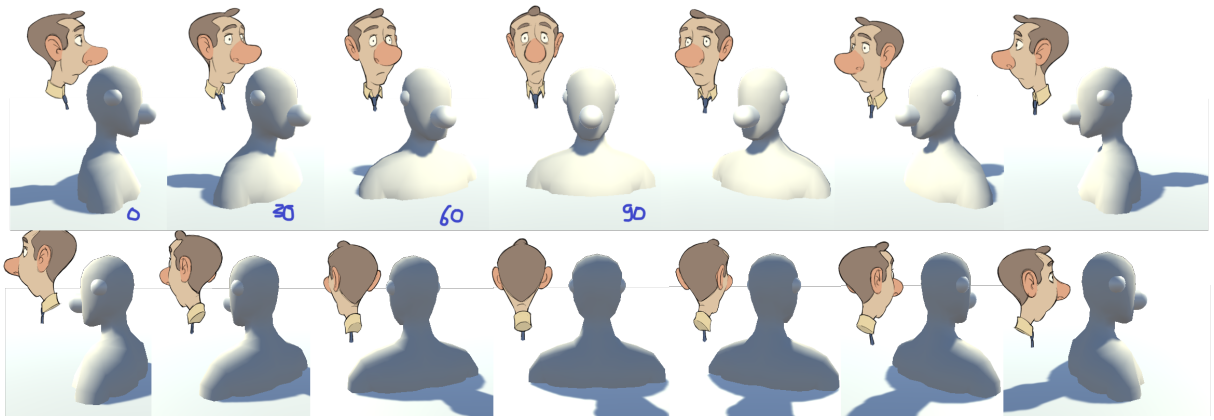


Figure 5.4: Images of overlapping turnaround and manikin images. This time for a different turnaround.


```
{
  "camera_angle_x": 0.6981317,
  "frames": [
    {
      "file_path": "./train/r_0",
      "rotation": 10.16641,
      "transform_matrix": [
        [ 0.0, -0.1240347, 0.9922777, 4.0 ],
        [ 0.9999999, -0.000000007393045, -0.0, 0.0 ],
        [ -0.0, 0.9922779, 0.1240347, 0.5 ],
        [ 0.0, 0.0, 0.0, 1.0]
      ]
    },
    {
      "file_path": "./train/r_1",
      "rotation": 10.16641,
      "transform_matrix": [
        [ 0.7071068, -0.08770581, 0.7016464, 2.828427 ],
        [ 0.7071068, 0.08770581, -0.7016464, -2.828427 ],
        [ -0.00000002252569, 0.9922779, 0.1240348, 0.5 ],
        [ 0.0, 0.0, 0.0, 1.0 ]
      ]
    },
    {
      "file_path": "./train/r_2",
      "rotation": 10.16641,
      "transform_matrix": [
        [ 1.0, 0.0, 0.00000005914438, 0.0000002384186 ],
        [ 0.00000005960465, 0.1240347, -0.9922779, -4.0 ],
        [ -0.0, 0.9922779, 0.1240347, 0.5 ],
        [ 0.0, 0.0, 0.0, 1.0 ]
      ]
    },
    {
      "file_path": "./train/r_6",
      "rotation": 10.16641,
      "transform_matrix": [
        [ -1.0, 0.0, 0.00000005914437, 0.0000002384186],
        [ 0.00000005960464, -0.1240347, 0.9922777, 4.0 ],
        [ 0.0, 0.9922779, 0.1240347, 0.5 ],
        [ 0.0, 0.0, 0.0, 1.0 ]
      ]
    },
    {
      "file_path": "./train/r_7",
      "rotation": 10.16641,
      "transform_matrix": [
        [ -0.7071066, -0.08770581, 0.7016464, 2.828428 ],
        [ 0.7071068, -0.08770577, 0.7016461, 2.828427 ],
        [ -0.00000001501713, 0.9922779, 0.1240347, 0.5 ],
        [ 0.0, 0.0, 0.0, 1.0]
      ]
    }
  ]
}
```


Figure 5.5: Complete turnaround dataset example, called "Man" dataset. On the left, images taken from artists turnaround of dimension 400x400. On the right, the json file responsible on maintaining the information on the chosen camera extrinsics and field-of-view. We can see that each image has its relative camera transform in the json file. The rotation parameter is irrelevant for our purposes.

6 Assessing NeRF's and Plenoxels' Results in Challenging Scenarios

As mentioned in Background sections 2.8 and 2.9, NeRF and Plenoxels were meant to work with a high number of images. These algorithms are not magic unfortunately, they learn the 3D representation that best explains the training images, they will not generalize well if not enough images are used in the training process. However, as we saw in Chapter 5, we do not have this luxury since most turnarounds depict a drawn character only from 5 to 8 different views.

Furthermore, NeRF and Plenoxels work with relatively precise camera positions. In real-world scenarios, Structure-from-Motion algorithms are used to infer camera positions and field-of-view, and they do it with a reasonable degree of precision for NeRF and Plenoxels to work well. Yet again, we do not know such parameters. In Chapter 5 we presented a method to guess to the best of our ability such parameters, and suggested some future development for obtaining more plausible camera positions, but right now our guesses are far from perfect.

In this chapter, both issues will be discussed to assess if these novel view-synthesis algorithms even stand a chance within this challenging scenario imposed by the novel turnaround dataset. The first section will cover how well NeRF works when a low number of images is available. The second will assess how much NeRF is sensible to imprecise camera positions. Given the similarities between NeRF and Plenoxels, it is safe to suppose that considerations that work with the former will be valid also for the latter.

6.1 NeRF Results when Few Views are Available

The number of images necessary to reconstruct a scene to the requested degree of quality is not known a priory, but from NeRF's paper [16] we can infer that view-synthesis SoA result are reachable when at least 25 images of the scene are available. However, as we saw in Chapter 5, we do not have this luxury since most turnarounds depict a drawn character only from 5 to 8 different views. Therefore, NeRF has been trained on the same scene shown in the first half of Figure 6.1 , lowering after each run the number of views of the scene. Unsurprisingly, the PSNR lowers for fewer images, following the results achieved by the original paper [16]. However, NeRF was able to achieve acceptable scene reconstruction even with a very limited number of views. In the second half of Figure 6.1 we can see how NeRF reconstructs surprisingly well the scene at hand, albeit with some (to be expected) artifacts. This gives us hope for achieving an acceptable character reconstruction also with the novel turnaround datasets, even if the number of views is limited.

6.2 NeRF Results when Camera Position is not Precise

Looking at NeRF's paper [16], aside from the synthetic scenarios which work with ground truth camera position, in real-world scenarios Structure-from-Motion algorithms are used to infer camera positions and field-of-view. These algorithms infer camera positions with a reasonable degree of precision, sufficient for NeRF and Plenoxels to work well. As mentioned in Chapter 5, considering that in our case camera positions will often be estimated with a low amount of precision, we want to assess how much imprecise camera positions affect NeRF results.

Of course, how much the camera position precision is important will depend also on how much it is distant from the target, since small changes in position for very far cameras with small field-of-views might end up missing the target altogether. For tests, we modeled a typical scenario in which most turnarounds (but even most NeRF synthetic scenes) fall in. In this scenario, cameras are distant 4 meters from the object of interest, the object's height is around 1.8 meters, the filed-of-view is enough

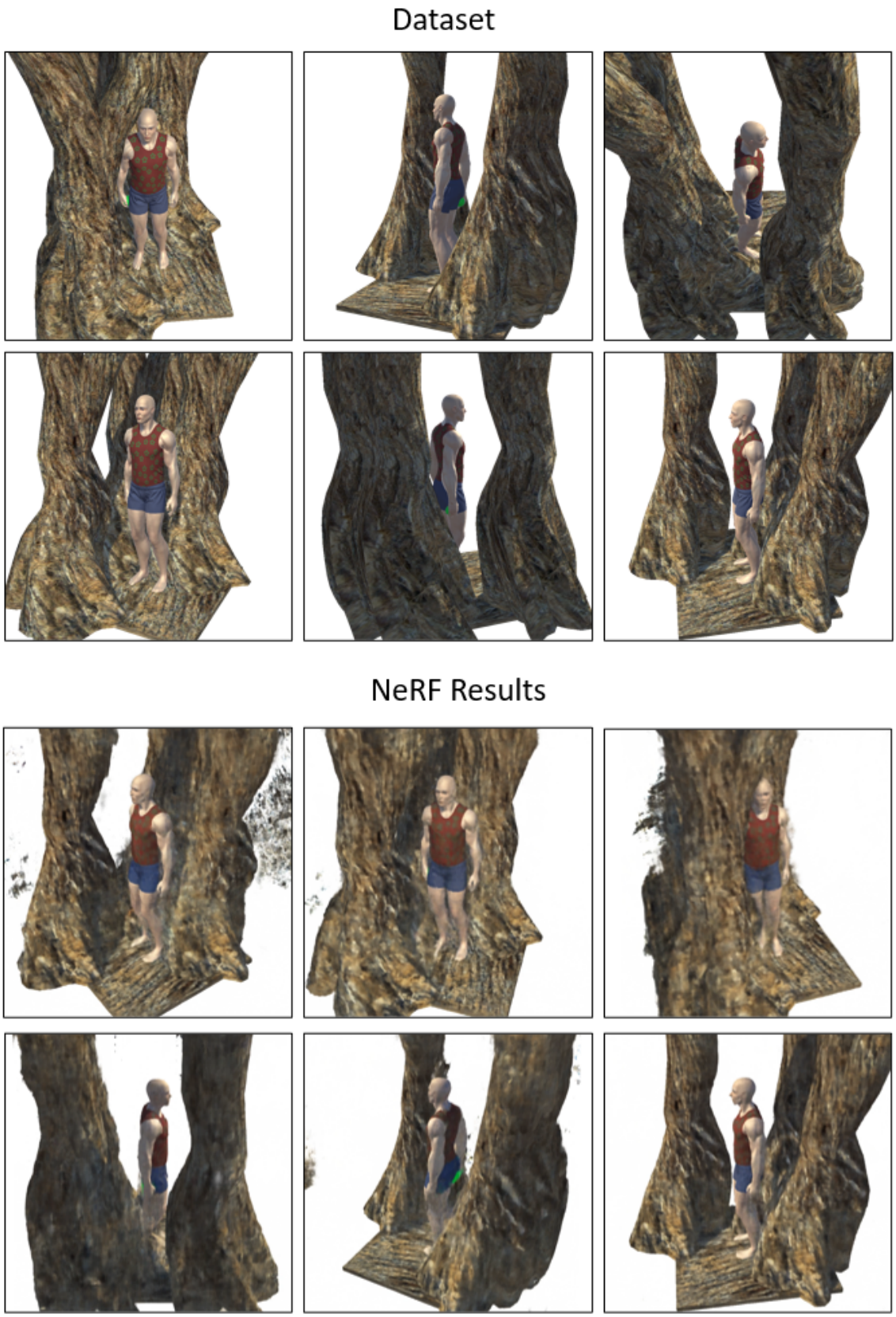


Figure 6.1: NeRF's performance when the number of views is limited. In this case, the training images available for understanding the scene at hand are just 6, shown in the upper half of the image. Rendering results of the trained NeRF are shown on the lower half. We can see that NeRF reconstructs surprisingly well the scene's structure and light in this challenging scenario, although with some imprecision and artifacts.

so that the target is well centered in the squared image.

We crate a dataset with such characteristics, capturing in this particular case a human warrior on a wooden floor. Target is captured disposing cameras around it spherically, from 100 different points of view. We want to know how much NeRF's reconstruction capabilities are affected giving to it always less precise camera positions. We start from ground truth camera positions and decrease run after run the precision in camera positions. As we can see in Figure 6.2 reconstruction of the scene is achieved even with camera positions correct at the centimeter level, without any noticeable quality reduction. However, reducing furthermore the camera positions precision heavily affects the reconstruction.

In our turnaround position estimation process, it could be quite realistic to get camera positions wrong at the decimeter level. It is therefore always-more a good idea to use a NeRF implementation which could correct camera positions within its training process, such as BARF [10] or [26][6], so to avoid heavy artifacts given by incorrect camera position estimation.

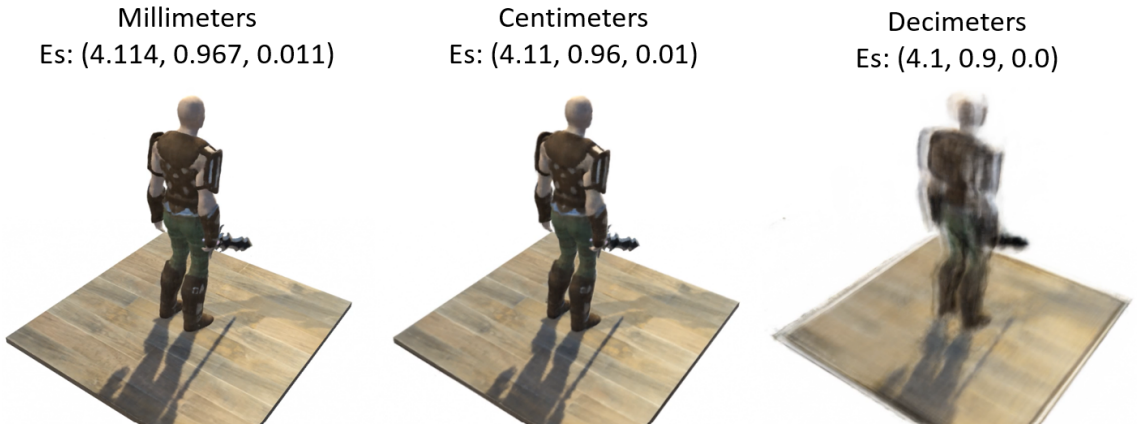


Figure 6.2: NeRF's performance when the camera position precision is decreased for different runs. As we can observe in this scenario, extreme precision in camera position is not needed. Reconstruction of the scene is achieved even with camera positions correct at the centimeter level, without any noticeable quality reduction. However, reducing furthermore the camera positions precision heavily affects the reconstruction (rightmost image).

7 Results

In this chapter are discussed and shown results obtained by training NeRF and Plenoxels on the novel turnaround datasets created. The first section presents the experimental setup used to create the datasets and the one used to train NeRF and Plenoxels. The second section focuses on the qualitative results obtained by NeRF on the novel datasets. The third section focuses on qualitative results obtained by Plenoxels on those same datasets. Finally, results achieved by turning NeRF representations into meshes are shown.

7.1 Experimental Setup

For the dataset creation process my laptop was used, which features astonishingly unremarkable specs. Any device able to run Unity will do. The presented plugin will work with Unity 2018.1 or later, it hasn't been tested on older versions. For training NeRF however, a GPU with a reasonable amount of memory (i.e. at least 6 GB of memory) is required, as NeRF is notoriously GPU memory hungry. Such constraint is less severe for Plenoxels. Therefore, an Nvidia RTX 2070 Ti with 8 GB of GPU memory is used for the training. The operating system used is Ubuntu 20.04 with CUDA 10.1 installed. For NeRF, its original tensorflow implementation is used. Also for Plenoxels the implementation used is the one given by its authors.

It has to be noted that with the exact same parameters, while using some of the most popular pytorch re-implementations, NeRF has not produced any results (white background image). The reason for this is unknown by the author, which therefore suggests to stick with the original tensorflow implementation unless the user is willing to do some debugging.

For the mesh creation process colab was used. The colab script is the same provided by NeRF's authors on the original NeRF's github page.

7.2 NeRF Renders Results on Turnaround Datasets

Lets recall that "Man" and "Boy" datasets have 5 training images, while "Head" has 14, as shown in Fig. 5.2. All datasets have been used to train NeRF until iteration 150000 was reached. In general, NeRF tends to converge to a good solution when this iteration is reached, and this with many more images to work on (i.e. 500). Infact, given our small datasets with a couple of images, NeRF did not show significant differences between iteration 150000 and 50000, indicating that convergence is reached already at iteration 50000. All datasets have been trained with $Nsamples = 64$ and $Nimportance = 128$, i.e. samples along rays are 64, if anything is found in between these samples than further 128 samples are used. Trainings lasted around 2 to 3 hours to reach iteration 50000, although keep in mind that training time largely depends on number of images and resolution. Results achieved by NeRF on "Head", "Man" and "Boy" datasets are shown in figures from Fig. 7.2 to Fig.7.8. Hereafter results are presented by grouping them in a dataset-specific manner, analyzing them independently in their own subsection.

7.2.1 NeRF Results on "Head" Dataset

Figure 7.2 shows the resulting renders from the Neural Radiance Field trained on "Head" dataset (shown in figure 5.2). Figure 7.3 shows renders of the head from the same network, but rendered from an angle, i.e. renders do not capture the target from straight forward but slightly from the top, with a 30 degree angle.

The "Head" dataset is, among the three datasets tested, the one with the most images, since the turnaround permitted it. For this reason, we do not find much overfitting (respect to the other datasets, as we'll see), i.e. rendered images look good all around the target and not only from views rendered exactly from positions that were present in training. This is because NeRF is trying to

explain the same volume and radiance from enough different points of view, and to do so it is forced to generalize. This also avoids the creation of floating artifacts as we will see for other results. Looking at results in Fig. 7.2 NeRF found a decent volume density and radiance values to explain the scene. Anyhow, results are far from perfect. Renders between close views often appear inconsistent, as if the head almost morphs and translates a little from one view to the other. Also, some high resolution detail fails to be reconstructed properly, as for the eyes and mouth. This issue is not caused by NeRF, but by the nature of the dataset. Even if the author drawing the turnaround has done an exceptional job by maintaining volume consistency over the whole turnaround, small changes in-between views occur often, especially considering this turnaround wasn't created with this purpose in mind. Most importantly, human errors have been surely made also when turning the transparent turnaround into separate images. The user has to crop and center perfectly the character into the center of the images, so as if the images were taken turning around the actual character that remains still. This is actually extremely challenging to do, aside from time consuming. Furthermore, lets remember that we have far from perfect camera positions explaining the scene, even if the Unity plugin has helped to tackle this issue. As mentioned in Chapter 6 a NeRF implementation that can correct camera positions in the training process, such as BARF [10], would probably be very helpful to counter these imperfections, that do appear in all datasets.

Looking at Fig. 7.3 we can see that renders done at an angle different to the one seen in training produces lower quality results. Colors are present, however they appear of a different shade, and strange white circles appear around the top of the head. The reason why these white circles appear is unknown to the author. Of course, this lowering in quality is unsurprising, since NeRF has no information about what the top of the head looks like, if not what it could infer by looking at the target horizontally. However, the resulting renders capture some color, even if with a slight difference in shade, and its surprising to see that there indeed is enough information to reconstruct the volume of the top of the head correctly. Having a top view of the head would have probably allowed for better results, however estimating the position of such view might not be that straight forward.

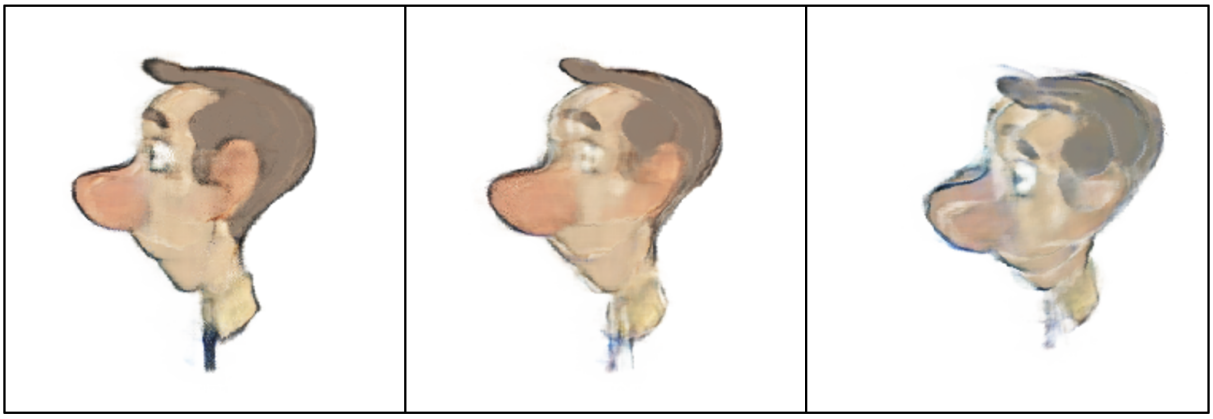


Figure 7.1: Here are gathered some relevant renderings from NeRF among the different ones produced on the "Head" dataset. The leftmost render is the best among the three since there was a training image in that very place. NeRF therefore exactly knows what to replicate from that view. This is not true for the other two renders, in which we can see a lower quality the more we tend to render further away from that sampled point, where we have less information of it.

Figure 7.2: NeRF on "Head" dataset. Renders depict the character horizontally (not at an angle). When rendering horizontally NeRF is able to better interpolate color values, since the turnarounds where drawn with this angle too.

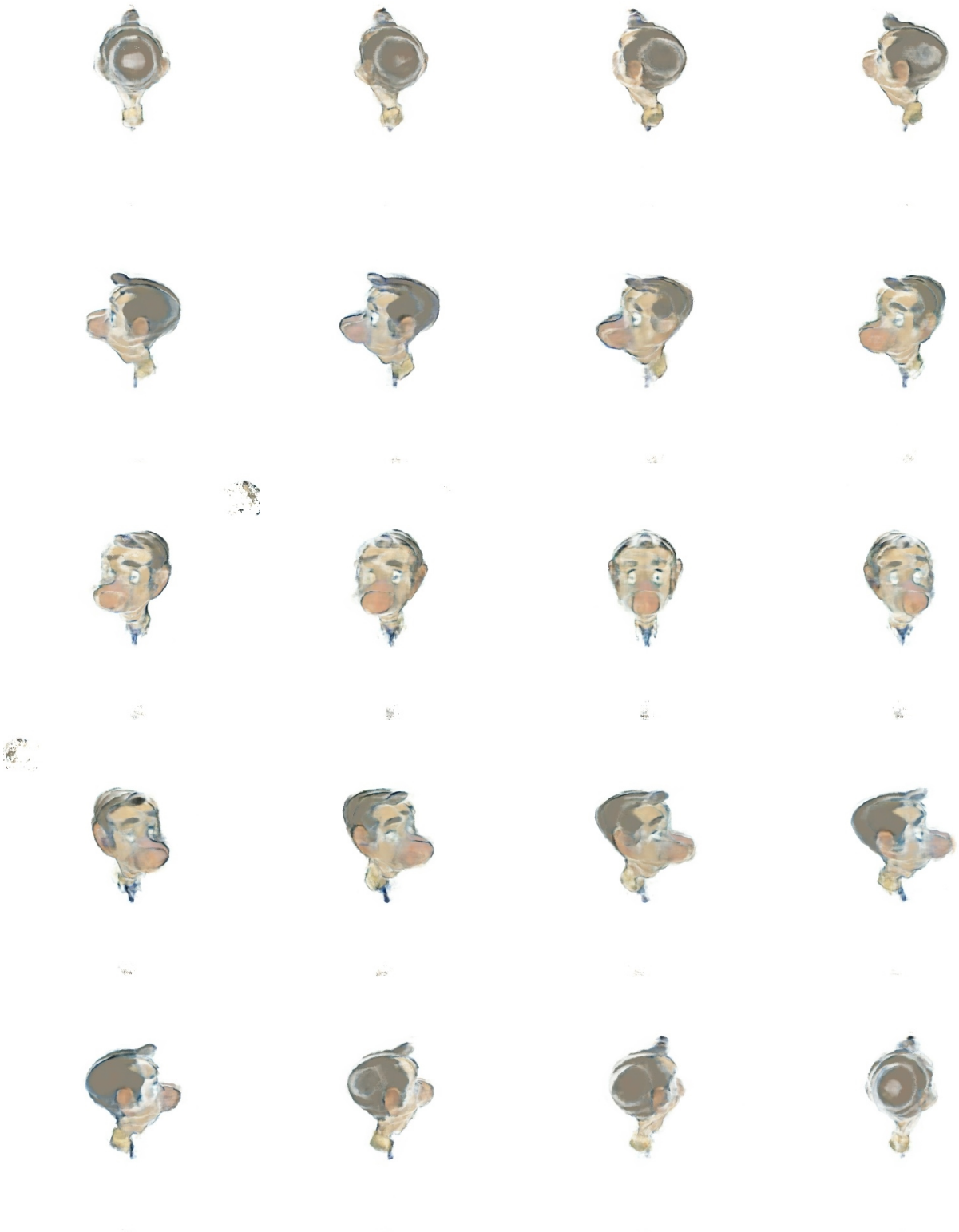


Figure 7.3: NeRF on "Head" dataset. Views capture the target at a 30 degree angle. Since the notwork has no information about what the head looks like from above, results appear qualitatively worse than ones rendered horizontally. If turnarounds depicted the character also from above, better results would be expected.

7.2.2 NeRF Results on "Man" Dataset

Results for the "Man" dataset (dataset shown in Fig. 5.5) are presented in Fig. 7.5. Images are rendered with a 10 degree angle w.r.t. the horizontal plane, while Fig. 7.6 renders the character from a more aggressive top-down angle of 60 degrees.

For this dataset, we can observe in fig. 7.5 that renders of the character's right side are qualitatively much better with respect to the renders of the left side. This strong difference in quality is explained by looking at the images in the dataset in Fig. 5.5, the turnaround has only views of the right side. NeRF will reconstruct characters left side thanks to information from front and back views, but of course the lower amount of information about this region of space appears clearly in the renders, especially for estimating plausible radiance values.

In this dataset it is evident that there is some overfitting. Looking at Fig. 7.5 some of the images have been trained very close with respect to the training images (positions are not quite the same since we are rendering at a slight 10 degree angle w.r.t. the horizontal plane). It is evident that results that are closest to the training ones appear cleaner and sharper with respect to the other renders, indicating that we are indeed reconstructing a volume and radiance that best explain the training images, but is unable to generalize that quality of rendering when rendering from different angles. However, even if the quality is inferior, renders from different angles angles produce decent results, showing us that NeRF at least understood that there is a compact volume before it and radiance values are interpolated decently.

These results show us that for decent results when rendering the target, a turnaround should at least count on eight different images, capturing the target from all sides.

Fig. 7.6 shows renders of the character with a more aggressive top-down angle of 60 degrees, a view that the network was not trained on. We indeed see that the lack of information about the top of the character shows shows, since radiance values are not correctly estimated, and therefore remain white. However, the volume of the hat was learned by the network.

Figure 7.4: Some relevant renderings among the ones produced on the "Man" dataset by NeRF. We can clearly see the difference in quality between the leftmost and the other two images. Since there is less information of our character from the left side and from the top, i.e. the turnaround does not cover that area, NeRF hallucinates mostly white colors, since there is no information to interpolate over from those directions.

Figure 7.5: NeRF on "Man" dataset. Views capture the target at a 10 degree angle. Colors and volume is guessed less accurately in some renders since the left's side of the man does not appear in the dataset.

Figure 7.6: NeRF on "Man" dataset. Views capture the target at a 60 degree angle. As for the head dataset, when rendering from the top, given the absence of information, NeRF is unable to interpolate colors accurately.

7.2.3 NeRF Results on "Boy" Dataset

Renders relative to the "Boy" dataset (shown in figure 5.2) can be found in Fig. 7.8.

Renders from this dataset are the worst among the three datasets tested. We can see that overfitting, such as floating artifacts, is much more present in this case. Again, in view-synthesis overfitting can be seen as rendering really well our subjects appearance, but only from the views on which it has been trained on. Indeed we can see again that renders similar to the ones seen in the dataset (Fig. 5.2) reconstruct the appearance of the character almost perfectly. However, when rendering from novel views, the quality of the render is much worse than expected. This is specifically shown in fig.7.7.

Overfitting in this case is much more severe, since NeRF is inferring the presence of parts of our character right in front of each camera or view, and not in the center of the scene as we would expect it to be. This this is why the character is correctly rendered from views present in training, but a really small variation in position shows the artifact. To counter this problem, the dataset can be expanded by using the same training image more than once, but with slightly modified extrinsics. Since the appearance of our character will not change significantly with slight movements of the camera, we can use this to our advantage to counter the aforementioned issue. This solution has been tested in the following Plenoxels section for the "Man" dataset, showing positive results. For lack of time such solution has not been implemented here.

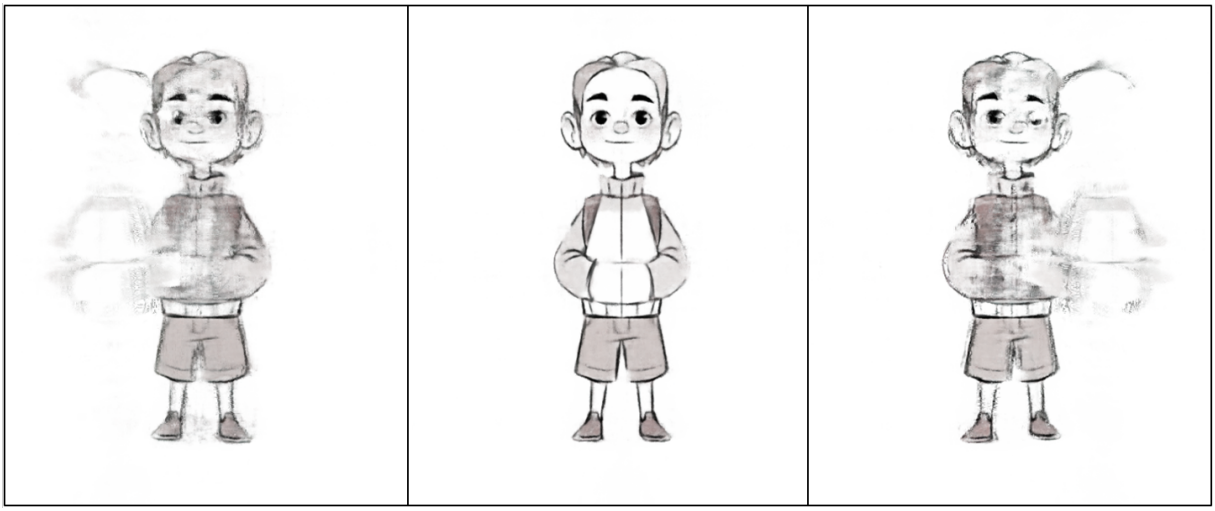


Figure 7.7: Some relevant renderings among the ones produced on the boy dataset by NeRF. Images show an artifact that can appear training NeRF when few views of the character are available. Leftmost and rightmost images show artifacts which appear to float right in front of the camera, and seem to alienate perfectly with our character when viewed from the center. Since there are few views that depict our character frontally NeRF hallucinates points right in front of the camera, since there is no reason why NeRF should place those points close to the center of the volume.

Figure 7.8: NeRF on "Boy" dataset. Views capture the target horizontally. Among the different datasets, this is the one over which NeRF generalizes less. We can indeed see that images close to the ones in training (Fig. 5.2) reconstruct the character much better w.r.t. the rest.

7.3 Plenoxels Renders Results on Turnaround Datasets

Results for Plenoxels have been obtained by running the original implementation of the paper. Training has been done by keeping default parameters. Training lasted around 5 minutes for both "Man" and "Head" datasets, which is a great advantage over the three hours required by NeRF. Results achieved by Plenoxels on "Head" and "Man" datasets are shown in figures from Fig. 7.10 to Fig.7.13. Hereafter, just as for NeRF, results are presented by grouping them in a dataset-specific manner, analyzing them independently in their own subsection.

7.3.1 Plenoxels Results on "Head" Dataset

Fig. 7.9 and 7.10 show the resulting renders from the Plenoxels architecture trained on the "Head" dataset (shown in figure 5.2). Renders capture the target from varying angles.

We can observe, especially from Fig. 7.9, that renders appear more noisy. Instead of appearing more like a drawing, renders appear as a point cloud, this is especially evident when rendering from above. It may be possible by changing parameters to impose on the scene a more dense plenoxel grid to avoid this point cloud effect. When rendered from above, results are qualitatively worse for the same reasons we described in NeRF, this shows especially in 7.9. This is because for the top of the head we do not have as much information as we do for the front and sides. However, even if the results from above are more noisy, radiance is captured with a higher fidelity with respect to NeRF, and no white rings appear on the top of the head. We can also observe significant overfitting. Although the head appearance is maintained, and no floating artifacts are present, points of view which where present in the training process are rendered exceptionally well with respect from the ones that weren't (fig. 7.9 shows clearly).

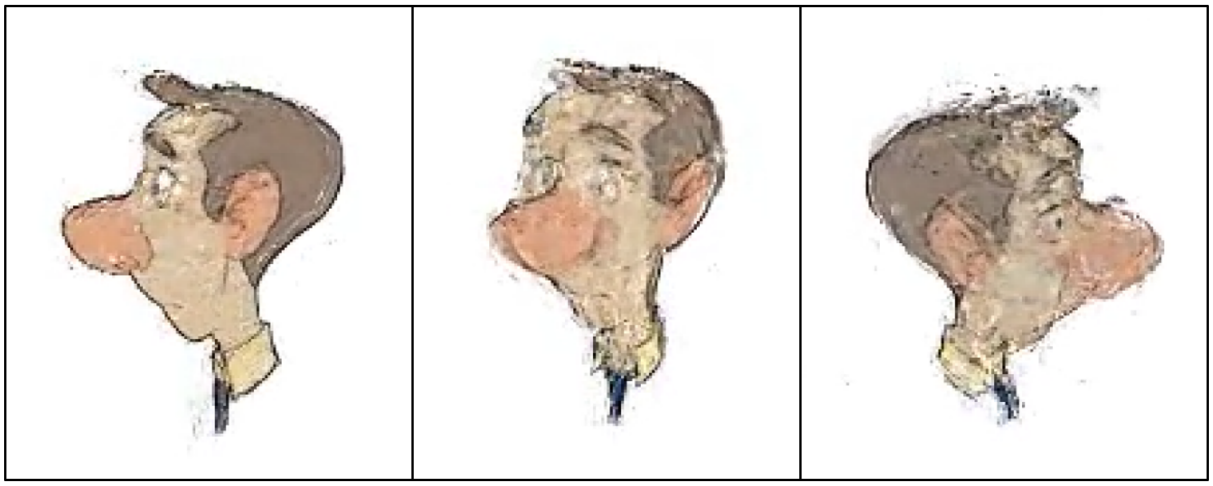


Figure 7.9: Here are gathered some relevant renderings from Plenoxels among the different ones produced on the "Head" dataset. The leftmost render is the best among the three since there was a training image in that very place. Plenoxels therefore exactly knows what to replicate from that view. This is not true for the other two renders, in which we can see a lower quality the more we tend to render from the top of our character, where we have less information of it.

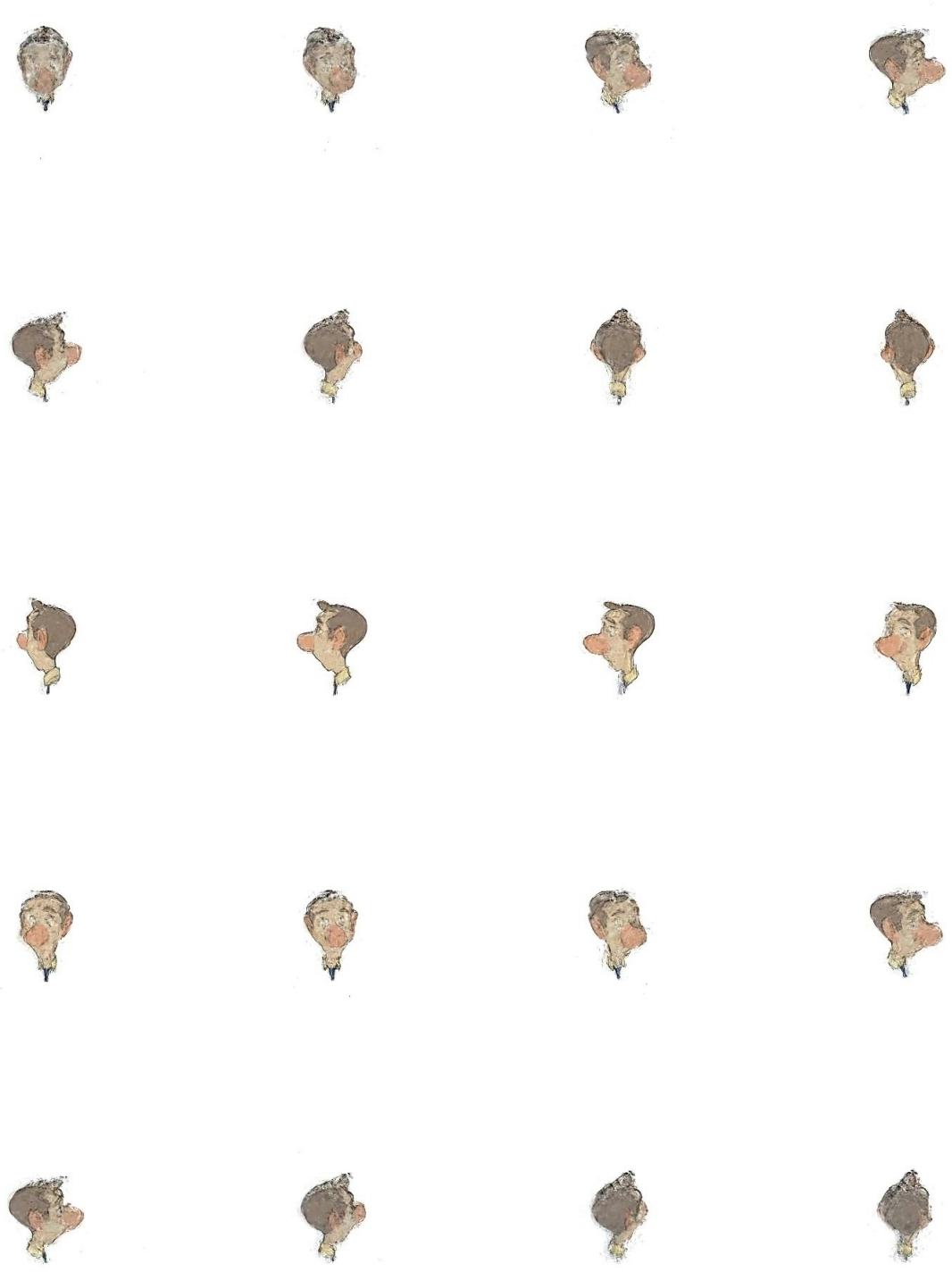
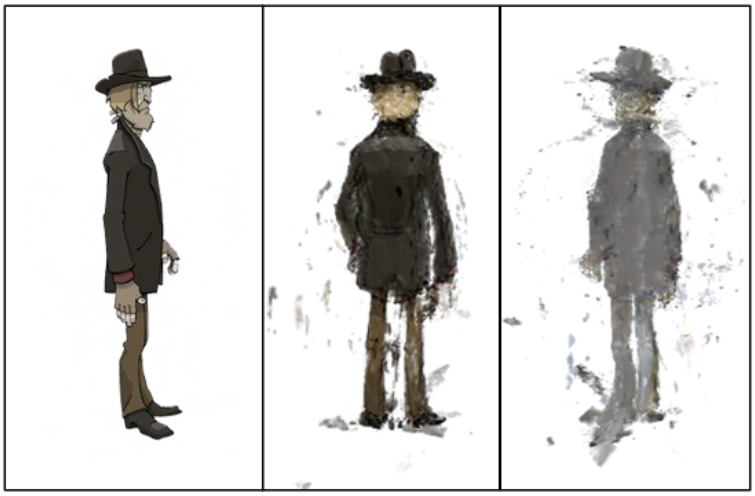


Figure 7.10: Plenoxels on "Boy" dataset. Renders capture the target from various heights and angles.

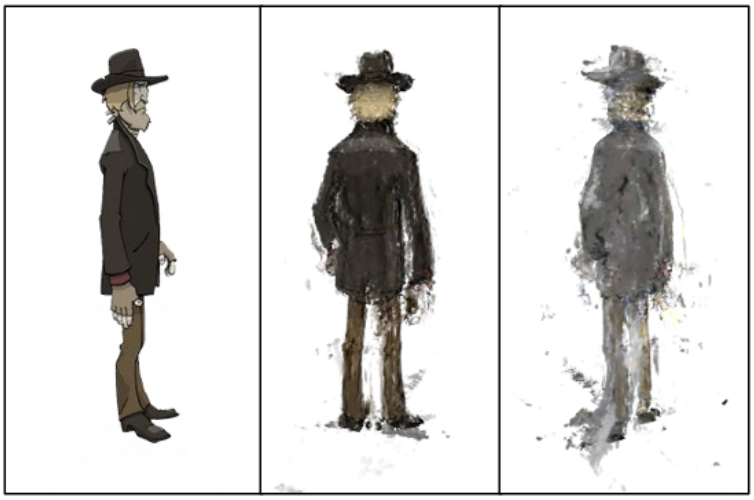
7.3.2 Plenoxels Results on "Man" Dataset

Results on the "Man" dataset (shown in Fig. 5.5) are presented in Fig. 7.13 rendered from varying angles. Improved results, due to a dataset modification detailed in section 5.4, are presented in Fig. 7.13.

First thing noticeable looking at results in 7.12 is the strong presence of floating artifacts. This is due to the too limited number of views of our dataset, which doesn't allow the plenoxel representation to generalize well for other views. NeRF suffered of the same issue when working on the "Boy" dataset. Renderings reconstruct the target perfectly when viewed from positions which were present in training, as seen in Fig. 7.11(a) but actually strong artifacts appear in the scene when rendering from other points of view, as in Fig. 7.11(b). Furthermore, when viewing the character from points of view that were not seen during training (i.e. sampled insufficiently) radiance values are not set properly, and are given a default white value, as can be seen in Fig. 7.11(c).



(a)



(b)

Figure 7.11: Here are gathered the most relevant renderings among two different runs of Plenoxels. The first run (a) used the standard "Man" dataset, as presented in 5.5. The second run (b) used a slightly modified version of this dataset, tweaked as described in section 5.4. We can see how, although views on which we trained on are still rendered perfectly (leftmost images), we drastically decrease the amount of floating artifacts in other views.

Figure 7.12: Plenoxels on "Man" dataset. Renders capture the target from various heights and angles.

Figure 7.13: Plenoxels on "Man" dataset. However, the dataset has been slightly modified to counter overfitting artifacts. Views capture the target from various heights and angles.

7.4 Mesh Generation Results

As mentioned when introducing NeRF in section 2.8, NeRF does not reconstruct the scene as a mesh. Instead, it uses continuous points in a bounded space that store learned density and color values, which then uses to produce images by using standard volume rendering techniques. To obtain a mesh of the character we need to convert the volumetric representation learned by NeRF into a mesh. To understand which parts of the scene are occupied and which are not, the 3D cubic space is discredited. The more we divide the space into cubes, the more detailed the resulting mesh. We choose weather a small cube is occupied or not by checking the density values of the continuous points inside it learned by NeRF. If the mean of the density of these points is higher than a user-chosen threshold, then the cube will be marked as occupied. We turn this resulting voxelized representation into a mesh by using the marching cubes algorithm [12], as suggested in [16]. Fig. 7.14 and 7.15 present the resulting meshes of such conversion.

Resulting meshes show that NeRF has indeed reconstructed a 3D representation out of the 2D turnaround dataset, and we successfully turned such representation into a usable mesh. The head and man 3D meshes however show significant artifacts, such as holes, duplicated body parts and general non-smoothness.

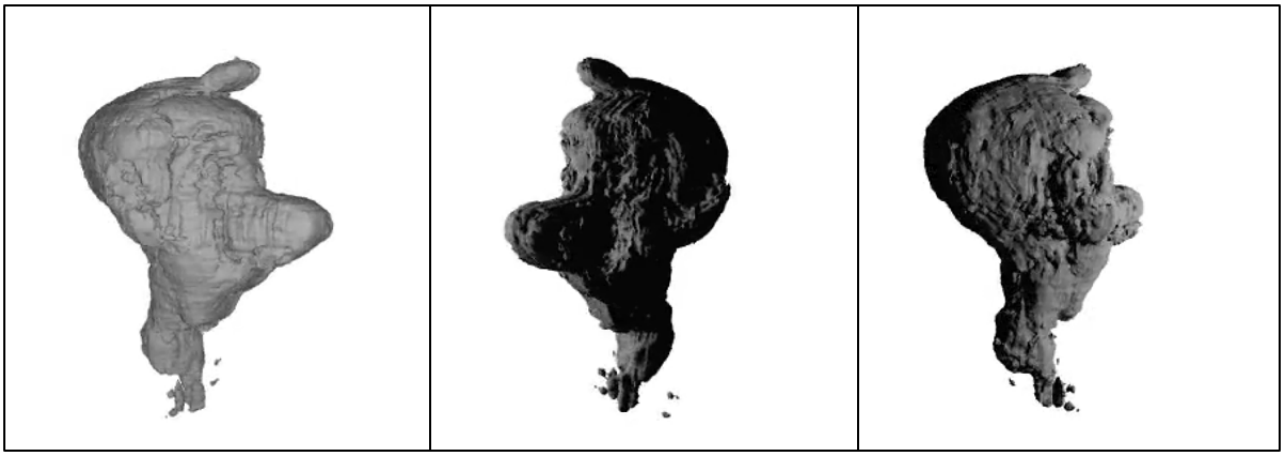


Figure 7.14: Mesh generated by applying marching cubes to NeRF's 3D reconstruction of the "Head" dataset.

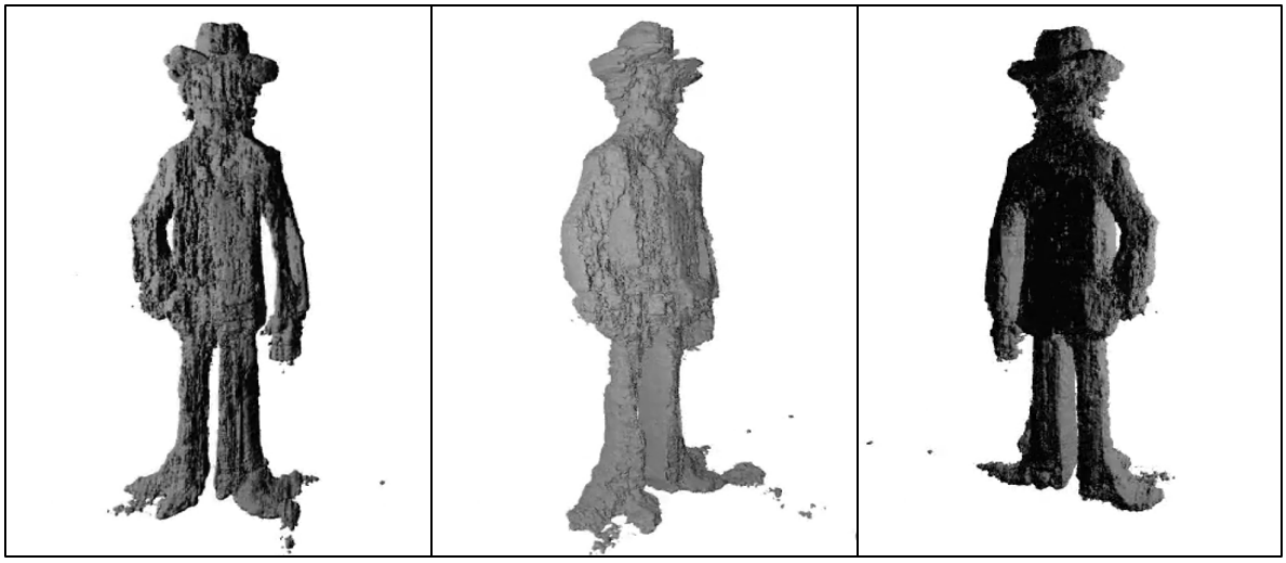


Figure 7.15: Mesh generated by applying marching cubes to NeRF's 3D reconstruction of the "Man" dataset.

8 Conclusions

This thesis addressed the challenging problem of reconstructing a 3D character representation out of its turnaround. The problem has been formulated in a view-synthesis manner, using SoA algorithms from this field - such as NeRF and Plenoxels - in order to render the character from novel views. The use of character turnarounds as datasets for their 3D reconstruction is a novelty, and as such the creation process of the dataset was a relevant part of this theses. A Unity plugin to help in the dataset creation process has been implemented and presented, and the process to create viable turnaround datasets has been thoroughly detailed. A short assessment was performed over NeRF's performance in challenging scenarios, in order to see if these algorithms even had a chance of producing viable results with few views and imprecise camera parameters available, which is the scenario imposed by the turnaround dataset. The assessment showed that even with few views NeRF can understand the 3D structure of the scene fairly well, however, this is true if views are sparse and camera positions are precise enough. Results over NeRF and Plenoxels were reported and detailed, showing that often these algorithms are indeed able to reconstruct a 3D representation of the 2D characters, however not without significant artifacts in the resulting renders, as one could expect from the initial assessment. Therefore, results show that this reconstruction approach is still not valuable for most applications, such as helping artists in the areas of digital animation or 3D modeling. However, this is a first attempt and the algorithms used were not meant for this challenging use case in mind. Therefore, there is a high chance that this might change with future improvements.

8.1 Limitations and Future Work

As mentioned previously, the proposed approach for reconstructing a 3D character representation out of its turnaround has severe limitations. Results show that, although a 3D representation of the 2D character can be learned, renders come with significant artifacts. The reasons why this happens, aside from possible drawing inconsistencies of the character by the artists, are mainly two: few views and imprecise camera position estimation (i.e. if the character was drawn as seen by the artist in the real world, where would the artist be placed with respect to the character?).

For solving the latter, the author has implemented a Unity plugin for better estimation of correct camera parameters. However, this still does not the problem to a satisfactory degree, since the process still needs the intervention of the user, the process is time consuming and there is no guarantee the user will choose precise and correct camera parameters. To build upon this solution, it would probably help to use a different implementation of NeRF which is able to slightly adjust camera positions during training. In essence, we want to be able to include in the differentiable training also the camera position estimation process, starting from the above initial guesses if possible. Projects which tackle camera extrinsics (pose) optimization in NeRF already exist [26][10][6], and provide usable implementations in their relative GitHub pages. This thesis has not tested these NeRF implementations, yet hopefully they may improve results.

Lets now address issues caused by the limited number of views available in turnarounds. With few views available renders of the character from unseen points of view are heavily affected by artifacts. Unfortunately, a solution to this issue is not as easy as capturing more images of a scene, since good turnarounds require time, effort and know-how for being created by artists. This is also the reason why no quantitative results were available, since in order to evaluate the quality of a render we would need other turnaround views of our character. Therefore, if good results are desired, the turnaround should feature more views of the character (even taken from atop). A more realistic solution would be to find a way to give a-priory knowledge of the drawn character to the algorithm, e.g. the character is human or its skin color does not depend on viewing angle. This is challenging, but surely an open area out of which the proposed approach would benefit greatly.

A possible improvement to the mesh could be to apply colors to it. Coloring would exploit the color information of the turnarounds. There are already some working algorithms made for this purpose, among whom the best results have probably been achieved by [21], which implemented this feature as a follow-up of his NeRF implementation. This "mesh colorization" algorithm has not been tested since no valuable results could be obtained with Chen's implementation of NeRF, just like with the NeRF pytorch based re-implementation which it was built upon [29]. However, adapting such algorithm to the NeRF results format of the original implementation should work.

Another improvement to the mesh as an end result for artists and creators could be an automatic rigging process. Since we have multiple views of our character, pose estimation could be performed on the different drawings of humanoid characters. Union of such poses may not be an easy task however.

Acknowledgements

I'd like to thank my supervisor Prof. Nicola Conci and co-supervisor Zeno Sambugaro for the many Zoom calls, for all the help and support provided and for the freedom given over how to develop this thesis.

Thanks to the sponsors, my parents. Finally, thanks to my dear brother Lorenzo, to my love Lara, to Mike and Lucia, to Luca, Francy and Francesca, to Giovanni and Ilaria and to Michele. Thanks for these wonderful years together.

Bibliography

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

[2] Marek Dvorožňák, Daniel Sýkora, Cassidy Curtis, Brian Curless, Olga Sorkine-Hornung, and David Salesin. Monster Mash: A single-view approach to casual 3d modeling and animation. *ACM Transactions on Graphics*, 39(6), 2020.

[3] S. M. Ali Eslami, Danilo Jimenez Rezende, Frederic Besse, Fabio Viola, Ari S. Morcos, Marta Garnelo, Avraham Ruderman, Andrei A. Rusu, Ivo Danihelka, Karol Gregor, David P. Reichert, Lars Buesing, Theophane Weber, Oriol Vinyals, Dan Rosenbaum, Neil Rabinowitz, Helen King, Chloe Hillier, Matt Botvinick, Daan Wierstra, Koray Kavukcuoglu, and Demis Hassabis. Neural scene representation and rendering. *Science*, 360(6394):1204–1210, 2018.

[4] Kyle Genova, Forrester Cole, Avneesh Sud, Aaron Sarna, and Thomas Funkhouser. Local deep implicit functions for 3d shape, 2020.

[5] Filip Hauptfleisch, Ondřej Texler, Aneta Texler, Jaroslav Křivánek, and Daniel Sýkora. StyleProp: Real-time example-based stylization of 3d models. *Computer Graphics Forum*, 39(7):575–586, 2020.

[6] Yoonwoo Jeong, Seokjun Ahn, Christopher Choy, Animashree Anandkumar, Minsu Cho, and Jaesik Park. Self-calibrating neural radiance fields. *CoRR*, abs/2108.13826, 2021.

[7] Chiyu "Max" Jiang, Avneesh Sud, Ameesh Makadia, Jingwei Huang, Matthias Niessner, and Thomas Funkhouser. Local implicit grid representations for 3d scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[8] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, page 143–150, New York, NY, USA, 1986.

[9] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. In *Ray Tracing Volume Densities*, SIGGRAPH '84, page 165–174, New York, NY, USA, 1984.

[10] Chen-Hsuan Lin, Wei-Chiu Ma, Antonio Torralba, and Simon Lucey. BARF: bundle-adjusting neural radiance fields. *CoRR*, abs/2104.06405, 2021.

[11] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *ACM Trans. Graph.*, 38(4), jul 2019.

[12] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, aug 1987.

[13] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[14] Moustafa Meshry, Dan B Goldman, Sameh Khamis, Hugues Hoppe, Rohit Pandey, Noah Snavely, and Ricardo Martin-Brualla. Neural rerendering in the wild, 2019.

[15] 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, 2019.

[16] 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.

[17] Thu Nguyen-Phuoc, Chuan Li, Lucas Theis, Christian Richardt, and Yong-Liang Yang. Hologan: Unsupervised learning of 3d representations from natural images, 2019.

[18] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision, 2020.

[19] Jeong Joon Park, Peter Florence, Julian Straub, Richard A. Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. *CoRR*, abs/1901.05103, 2019.

[20] Jeong Joon Park, Peter Florence, Julian Straub, Richard A. Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. *CoRR*, abs/1901.05103, 2019.

[21] Chen Quei-An. Nerf pl: a pytorch-lightning implementation of nerf, 2020.

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

[23] Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. *CoRR*, abs/1906.01618, 2019.

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

[25] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford, CA, USA, 1998. AAI9837162.

[26] Zirui Wang, Shangzhe Wu, Weidi Xie, Min Chen, and Victor Adrian Prisacariu. Nerf-: Neural radiance fields without known camera parameters. *CoRR*, abs/2102.07064, 2021.

[27] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, jun 1980.

[28] Zexiang Xu, Sai Bi, Kalyan Sunkavalli, Sunil Hadap, Hao Su, and Ravi Ramamoorthi. Deep view synthesis from sparse photometric images. *ACM Trans. Graph.*, 38(4), jul 2019.

[29] Lin Yen-Chen. Nerf-pytorch, 2020.

[30] Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks, 2021.

[31] Jun-Yan Zhu, Philipp Krähenbühl, Eli Shechtman, and Alexei A. Efros. Generative visual manipulation on the natural image manifold, 2018.