

# Ergebnisse SynthData

André Sers

Okttober 2023 - Januar 2024

## Inhalt

|   |    |
|---|----|
| 1. Übersicht .....                                      | 2  |
| 1.1. System .....                                       | 2  |
| 2. COLMAP .....   | 2  |
| 2.1. SIFT Feature extraction .....                      | 2  |
| 2.2. Matching .....                                     | 3  |
| 2.3. Gemoetrische Verifizierung .....                   | 3  |
| 2.4. Rekonstruktion .....                               | 3  |
| 2.5. Probleme .....                                     | 3  |
| 2.6. Nutzung von COLMAP .....                           | 5  |
| 2.7. Weitere Möglichkeiten zu Posenbestimmung .....     | 7  |
| 3. Instant-NGP .....                                    | 7  |
| 3.1. Mathematische Beschreibung von Szenen .....        | 7  |
| 3.2. Volume Rendering .....                             | 8  |
| 3.3. Mutliresolution Hashgrid encoding .....            | 8  |
| 3.4. Nutzung von Instant-NGP .....                      | 9  |
| 3.5. Anforderungen an Bilder für NeRF-Generierung ..... | 10 |
| 3.6. Ergebnisse .....                                   | 11 |
| 4. Nvdiffrc .....                                       | 13 |
| 4.1. Mathematische Beschreibung von Szenen .....        | 13 |
| 4.2. Rendering .....                                    | 14 |
| 4.3. Nutzung von nvdiffrc .....                         | 15 |
| 4.4. Anforderungen an Trainingsdatensatz .....          | 15 |
| 4.5. Ergebnisse .....                                   | 16 |
| 5. Nvdiffrcmc .....                                     | 20 |
| 5.1. Monte-Carlo Integration .....                      | 21 |
| 5.2. Open Image Denoiser .....                          | 21 |
| 5.3. Anforderungen an den Datensatz .....               | 22 |
| 6. Dokumentation Codebasis .....                        | 22 |
| 6.1. Instant-NGP .....                                  | 22 |
| 6.2. nvdiffrc .....                                     | 22 |
| Verweise .....  | 24 |

# 1. Übersicht

Das Ziel der Arbeit ist es, Methoden zu untersuchen, mit deren Hilfe der EIBA/MVIP Datensatz von alten elektronischen Geräten virtuell zu rekonstruieren, um daraus automatisiert synthetische Daten für das Training einer Klassifikations-KI generieren zu können. Dazu müssen die Methoden in der Lage sein, anhand von Bilddaten der jeweiligen Teile hochauflösende Meshes mit zugehöriger Textur und Material für eine weiter Simulation zu generieren. Im Laufe der Arbeit wurden drei Methoden untersucht, für zwei davon sollen im Folgenden die erzeugten Ergebnisse präsentiert werden. Die letzte der drei Methoden, nvdiffrcmc, wurde zum Abschluss dieser Dokumentation noch nicht ausführlich genug ausgewertet, um hier Ergebnisse präsentieren zu können.

## 1.1. System

Das verwendete System hat sich im Laufe der Arbeit verändert. Alle in dieser Dokumentation gezeigten Daten und Ergebnisse wurden jedoch auf einem System produziert:

```
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.6 LTS
Release:        20.04
Codename:       focal
GPU:           NVIDIA RTX A6000
VRAM:          49GB
```

# 2. COLMAP

Die Rekonstruktion der Objekt soll in unserem Fall anhand von einzelnen Bildern der Objekte, aufgenommen aus unterschiedlichen Perspektiven, geschehen. Damit das überhaupt möglich ist, benötigen wir Informationen über die Kameras, welche die einzelnen Bilder aufgenommen haben. Genauer, wir benötigen Positionen von virtuellen Kameras im dreidimensionalen Raum, welche in Kombination mit den Bildern die dreidimensionale Szene repräsentieren. COLMAP [1], [2] ist eine Pipeline, die genau diese Funktionalität implementiert. Die Grundlegende Vorgehensweise von COLMAP ist dabei die folgende:

## 2.1. SIFT Feature extraction

Der erste Schritt ist, aus jedem Bild  $I_i \in I$  eine Menge  $F_i = \{(x_j, f_j) \mid j = 1, \dots, N_{F_i}\}$  sogenannter lokaler Features an Position  $x_j \in R^2$  mit Beschreibungsvektor  $f_j$  zu extrahieren. SIFT [3] stellt dabei den Goldstandard in Bezug auf Robustheit der gewonnenen Features dar. Die extrahierten Features müssen invariant gegenüber geometrischer und radiometrischer Veränderungen sein, um in mehreren Bildern detektiert werden zu können.

SIFT geht dafür wie folgt vor: Zunächst wird auf die einzelnen Bilder in unterschiedlichen Auflösungen ein gaußscher Weichzeichner mit unterschiedlichen Stärken (Oktaven) angewandt. Sei eine Oktave  $L(x, y, \sigma)$ , wobei  $\sigma$  die Stärke des gaußschen Weichzeichners  $G(x, y, \sigma)$  ist. Anschließend wird für alle Bilder für je zwei Oktaven eines Bildes die Differenz zweier Oktave  $DoG = L(x, y, \sigma) - L(x, y, k\sigma)$ ,  $k \neq 1$  gebildet. In dieser Menge an Differenzbildern werden nun die eigentlichen Features entnommen. Dazu werden alle Pixel in  $DoG(k)$  betrachtet und der Laplace-Operator auf diesen Pixel und seine unmittelbaren Nachbarn, sowie die korrespondierenden Pixel in  $DoG(k+1)$  sowie  $DoG(k-1)$  angewandt. Entspricht dieser Pixel einem lokalen Extremum, so handelt es sich um einen Featurepunkt. Viele dieser Featurepunkte liegen entlang von Kanten, diese werden in einem nächsten Schritt entfernt, um die Robustheit zu erhöhen. Die Featurepunkte werden anschließend Positions- und Orientierungskodiert, sodass das Paar  $(x_j, f_j)$  entsteht.

## 2.2. Matching

Anschließend werden die Bilder mit korrespondierenden Featuremengen  $F_i$  paarweise nach dem nächsten Nachbarn Prinzip verglichen, um Paare an Bildern zu finden, welche die ähnlichsten Featuremengen haben. Der Output dieses Schrittes ist eine Menge an Bildpaaren  $\{\{I_a, I_b\} \mid I_a, I_b \in I, a < b\}$  sowie deren korrespondierende Featurematrix  $M_{ab} \in F_a \times F_b$ .

## 2.3. Gemoetrische Verifizierung

Da die Zuordnung der einzelnen Bildpaare nur auf Basis der jeweiligen Features geschieht, müssen die Zuordnungen verifiziert werden. Dazu wird versucht, eine Transformationsmatrix zu approximieren, welche die Features  $F_a$  auf die Features  $F_b$  abbildet. Das Ergebnis dieses Schrittes ist ein sogenannter Szenengraph, welcher als Knoten die Featurevektoren  $F_i$  und als Kanten die Transformationen  $M_{i,j}$  besitzt.

## 2.4. Rekonstruktion

Anschließend wird eine Menge  $P = \{P_c \in SE(3) \mid c = 1, \dots N_p\}$  für registrierte Bilder sowie die Rekonstruierte Szene als Menge von Punkten  $X = \{X_k \in R^3 \mid k = 1, \dots N_X\}$  generiert. Die von COLMAP erzeugte Rekonstruktion ist allerdings nicht genau genug, um für meine Zwecke ausreichend zu sein, weshalb ich nur an den Posen für die Rekonstruktion interessiert bin.  $P$  beinhaltet nebst der Kameraposen auch die approximierten intrinsischen Parameter der Kamera  $c$ .

## 2.5. Probleme

Die Nutzung von COLMAP kann im wesentlichen über drei verschiedene Anwendungen erfolgen:

- Vorkompilierte COLMAP Anwendung
- COLMAP conda Umgebung
- neu kompilierte COLMAP Anwendung

Für das von mir verwendete Linux basierte System, kommen nur die Conda Umgebung, sowie eine komplett neu kompilierte Version von COLMAP in Frage. Im Verlauf der Arbeit habe ich jedoch feststellen müssen, dass die Conda Umgebung auf meinem System teils unvorhergesehenes Verhalten aufweist und deshalb nicht für die Anwendung geeignet ist. Aus mir nicht erklärblichen Gründen versagt die Conda Umgebung bei der Rekonstruktion teilweise vollständig. Dadurch sind die generierten Kameraposen entweder unvollständig, es werden nicht alle Bilder aus dem Inputdatensatz in die Szene registriert, oder die Kameraposen weisen starke Fehler auf. Um dieses Problem zu verdeutlichen, hier folgende Daten:

```
BATCH1:
Elapsed time: 10.106 [minutes]
Cameras Camera(id=1, model='SIMPLE_RADIAL', width=4000, height=4000,
params=array([ 7.55378388e+03,  2.00000000e+03,  2.00000000e+03, -1.71299016e-01]))
Images # 66
Points (15589, 3) Visibility (15589, 66)
Depth stats 2.860853252038066 4.873137199631131 3.8783812706720386
```

```
BATCH2:
Elapsed time: 16.667 [minutes]
Cameras Camera(id=1, model='SIMPLE_RADIAL', width=4000, height=4000,
params=array([ 7.80111554e+03,  2.00000000e+03,  2.00000000e+03, -1.60177149e-01]))
Images # 66
Points (22942, 3) Visibility (22942, 66)
Depth stats 0.697014548133437 46.52244101720975 4.764108245684797
```

BATCH3:

```
Elapsed time: 11.601 [minutes]
Cameras Camera(id=1, model='SIMPLE_RADIAL', width=4000, height=4000,
params=array([ 7.62831290e+03,  2.00000000e+03,  2.00000000e+03, -1.56530234e-01]))
Images # 66
Points (18896, 3) Visibility (18896, 66)
Depth stats 0.6787799187114506 4.949889500772263 3.931092275389676
```

BATCH4:

```
Elapsed time: 18.514 [minutes]
Cameras Camera(id=1, model='SIMPLE_RADIAL', width=4000, height=4000,
params=array([ 7.73374786e+03,  2.00000000e+03,  2.00000000e+03, -1.32803609e-01]))
Images # 66
Points (20455, 3) Visibility (20455, 66)
Depth stats 1.8619167265626597e-05 40.67507052546553 5.239458383295887
```

BATCH5:

```
Elapsed time: 10.258 [minutes]
Cameras Camera(id=1, model='SIMPLE_RADIAL', width=4000, height=4000,
params=array([ 7.61870373e+03,  2.00000000e+03,  2.00000000e+03, -1.65583590e-01]))
Images # 66
Points (15786, 3) Visibility (15786, 66)
Depth stats 2.9081676986322034 5.027101458255294 4.040271527843994
```

Die oben gezeigten Ergebnisse sind ein Auszug aus dem Log mehrere, auf den gleichen Inputdaten durchgeföhrter Durchläufe von COLMAP. Um eine maximale Unabhängigkeit zwischen den Durchläufen zu garantieren, wurde die Arbeitsumgebung von COLMAP bei vor jedem Durchlauf komplett neu aufgesetzt und die Bilder neu in die Umgebung geladen. Zu sehen ist, dass die Rekonstruktion der Kameraparamenter und der Kameraposen sich zwischen den einzelnen Durchläufen teils stark unterscheidet. Dies hat auch unmittelbare Auswirkungen auf die Rekonstruktion der Objekte im weiteren Verlauf der Arbeit, hier ein Beispiel aus der Rekonstruktion des NVIDIA Datensatzes [4] mit Hilfe von nvdiffrc:

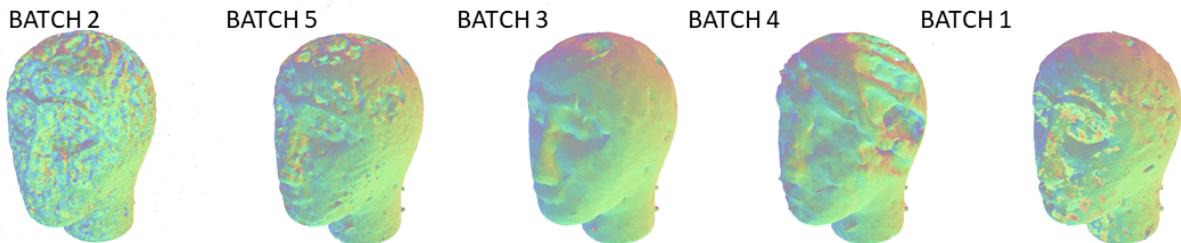


Figure 1: Rekonstruktionen des Datensets mit verschiedenen COLMAP Posen

Abhängig von der Qualität der Kameraposen weisen die Rekonstruktionen teils starke Fehler und Ungenauigkeiten auf, da nvdiffrc keine Korrektur der Kameraposen vorsieht. Führt man das gleiche Experiment mit der kompilierten Version von COLMAP durch, so lässt sich diese Verhalten nicht reproduzieren und die generierten Kameraposen sind wesentlich robuster.

Wie bereits schon kurz angesprochen, habe ich keine finale Erklärung, weshalb diese Fehler nur bei Nutzung der Conda Umgebung auftreten. Eine Möglichkeit ist, dass Conda keine dedizierte Nutzung der GPU des Systems erlaubt und insbesondere bei NVIDIA GPUs keine CUDA Unterstützung bereitstellt. Da insbesondere das SIFT-Feature extracting eine sehr rechenaufwendige Operation darstellt, kann es sein, dass das Verfahren ohne GPU Unterstützung schlicht nicht so präzise arbeiten kann wie mit der Unterstützung. Hinzu kommt noch, dass eine rein CPU gestützte Durchführung

von SIFT damit von der aktuellen Auslastung der CPU Abhängig ist, was die Unterschiede in zwischen den einzelnen Durchläufen erklären könnte. Allerdings kann ich mit der kompilierten Version von COLMAP auch bei absichtlicher Durchführung auf der CPU die Ergebnisse aus der Conda Umgebung nicht reproduzieren.

COLMAP bietet die Möglichkeit einer gesamten Rekonstruktion der Szene, welche aber für meine Zwecke nicht geeignet ist. Dies liegt an den gestellten Anforderungen an die Rekonstruktion. Für eine weitere Simultaion der Objekte benötige ich neben der Rekonstruktion des Objektes eine Rekonstruktion der Textur sowie des Materials. Sowohl Textur als auch Material werden von COLMAP bei der Rekonstruktion nicht generiert. In wie fern sich die von COLMAP erzeugte Rekonstruktion für die Generierung synthetischer Daten geeignet ist, wurde von mir während dieser Arbeit nicht weiter untersucht. COLMAP bietet desweiteren eine Möglichkeit der Visualisierung der generierten Kameraposen. Dies ermöglicht zwischen den Rekonstruktion eine verifizierung der Posen und einen eventuellen Vergleich. Zwar sind minimale Abweichungen der Posen nicht erkennbar, dennoch können fehlerhafte Generierungen sehr leicht ausgeschlossen werden:



Figure 2: Beispiel von COLMAP Kameravisualisierung, links erfolgreiche Rekonstruktion, rechts fehlerhafte Posen

Um die oben beschriebenen Probleme mit COLMAP zu beheben, benötigte ich eine kompilierte Version von COLMAP. Mit dieser Version traten die Schwankungen in den rekonstruierten Parametern nicht mehr auf, auch wenn sich an der Konfiguration der Pipeline im Vergleich zur Conda Umgebung nichts änderte. Die Kompilierung von COLMAP gestaltete sich auf meinem System äußerst schwierig, den genauen Ablauf lege ich in Abschnitt 2.6 dar.

## 2.6. Nutzung von COLMAP

### 2.6.1. Notwendige Pakete

```
sudo apt-get install \
    git \
    cmake \
    ninja-build \
    build-essential \
    libboost-program-options-dev \
    libboost-filesystem-dev \
    libboost-graph-dev \
    libboost-system-dev \
    libeigen3-dev \
    libflann-dev \
    libfreeimage-dev \
    libmetis-dev \
    libgoogle-glog-dev \
```

```
libgtest-dev \
libsdl2-dev \
libglew-dev \
qtbase5-dev \
libqt5opengl5-dev \
libcgal-dev \
```

### 2.6.2. Ceres

Folgt man den Entwicklern von COLMAP, so wird mit den notwendigen Paketen der ceres-solver installiert:

```
sudo apt-get install libceres-dev
```

Auf dem von mir verwendeten System wird durch diesen Befehl jedoch eine nicht funktionstüchtige Version von Ceres installiert, was im weiteren Verlauf der Kompilierung zu Problemen führt. Anstelle der direkten Installation von Ceres muss hier auf eine modifizierte Version zurückgegriffen werden, welche von dem GitHub Benutzer brentyi [5] zur Verfügung gestellt wird:

```
sudo apt-get remove libceres-dev # remove any broken ceres-solver installation
git clone https://github.com/brentyi/ceres-solver
cd ceres-solver
mkdir build
cd build
cmake .. -DBUILD_TESTING=OFF -DBUILD_EXAMPLES=OFF
make -j
sudo make install
```

### 2.6.3. Konfiguration von COLMAP

Einige der installierten Pakete sind entweder fehlerhaft, oder aber nicht auf die Installation auf multi-user Systemen ausgelegt. Anders als bei Ceres können diese Probleme allerdings durch eine Änderung der Systemkonfiguration behoben werden:

```
sudo ln -s /include /usr/include
```

Die Pakete *libqt5-dev*, *Qt5OpenGL* und *Qt5Widgets* existieren separat als Conda-Pakete. Für den Fall, dass parallel zur Kompilation von COLMAP diese Pakete via Conda installiert sind, müssen einige Referenzen auf diese Pakete angepasst werden. Dazu wird in *CMakeLists.txt* des COLMAP Verzeichnisses folgende Zeile hinzugefügt:

```
set(CMAKE_PREFIX_PATH "/usr/lib/x86_64-linux-gnu/cmake")
```

Um COLMAP mit CUDA-Unterstützung zu kompilieren, muss das cuda-toolkit installiert werden, sowie CUDA in den jeweiligen PATH der Kompilierung von COLMAP hinzugefügt werden:

```
sudo apt-get install nvidia-cuda-toolkit
sudo apt-get install nvidia-cuda-toolkit-gcc
export PATH="/usr/local/<your-cuda-version>/bin:$PATH"
```

Anschließend sollte das System für die Kompilierung von COLMAP konfiguriert sein

### 2.6.4. Kompilierung von COLMAP

```
git clone https://github.com/colmap/colmap.git
cd colmap
mkdir build
cd build
cmake .. -DCMAKE_CUDA_COMPILER=/usr/local/<your-cuda-version>/bin/nvcc>
-DCMAKE_CUDA_ARCHITECTURES=native -GNinja
```

```
ninja
sudo ninja install
```

Falls COLMAP ohne CUDA-Unterstützung kompiliert werden soll, ändert sich der Ablauf zu:

```
git clone https://github.com/colmap/colmap.git
cd colmap
mkdir build
cd build
cmake .. -GNinja
ninja
sudo ninja install
```

Damit COLMAP erfolgreich Kameraposen aus Bildern erstellen kann, müssen einige Anforderungen an die Bilder gestellt sein. Da diese Anforderungen jedoch im wesentlichen mit den Anforderungen für Instant-NGP bzw. nvdiffrc übereinstimmen, werde ich diese in den jeweiligen Abschnitten diskutieren.

## 2.7. Weitere Möglichkeiten zu Posenbestimmung

COLMAP stellt nicht die einzige Möglichkeit dar, Kameraposen für Fotogrammetrieverfahren zu generieren. Eine weitere, recht unkomplizierte, Methode ist die Generierung mittels sogenannter ArUco-Marker. ArUco-Marker sind künstlich erstellte Muster aus schwarzen und weißen Kästchen festgelegter Größe, welche von Computern sehr einfach in Bildern identifiziert werden können. Auf Grund der bekannten Abmessungen lassen sich anhand dieser Marker Position und Parameter der Kamera leicht bestimmen. Der Grund, warum ich mich gegen die Nutzung von ArUco Markern zur Posengenerierung entschieden habe ist, dass alle von mir verwendeten Methoden Anforderungen an das Format der Kameraposen stellen, die sehr nah an das schon vorhandene Format des Outputs von COLMAP herankommen. Außerdem verwenden die Methoden und COLMAP eine unkonventionelle Form von Koordinatensystemen für die Transformation von Punkten, die auf Grund fehlender Dokumentation sowohl auf Seiten von COLMAP als auch auf Seiten von Instant-NGP/nvdiffrc nur sehr schwer nachvollziehbar ist. Der Übersichtlichkeit halber und auch um bei eventuellen Transformationen von Koordinatensystemen keine Fehler einzubauen, die das Gesamtergebnis beeinflussen, habe ich mich dazu entschieden, COLMAP anstelle von ArUco-Markern zu nutzen.

## 3. Instant-NGP

NVIDIA bietet mit Instant-NGP [6] eine sehr aktuelle Möglichkeit, dreidimensionale Szenen aus einer Reihe von Bildern, die die gewünschte Szene aus verschiedenen Perspektive abbilden, zu rekonstruieren. Die Basis für diese Rekonstruktion stellen dabei sogenannte Neural Radiance Fields (kurz NeRF) [7] dar. NeRFs sind eine recht neue Methode zur Abbildung dreidimensionaler Szenen. 3D-Szenen werden typischerweise in Form von Pixelrastern oder Polygonmeshes repräsentiert. Diese Formen der Darstellung haben jedoch das Problem, dass eine hohe Detailsauflösung mit einem starken Anstieg an Speicherbedarf verbunden ist. Hinzu kommt noch, dass Meshes Probleme haben, nicht solide Oberflächen akkurat darzustellen und die Generierung dieser Meshes oft zeit- und ressourcenaufwändig ist.

NeRFs hingegen stellen neuronale Netzwerke dar, welche die entsprechende Szene repräsentieren. Dazu werden die Netzwerke so trainiert, dass für alle Punkte der repräsentierten Szene Farb- und Dichtewerte vorhergesagt werden. Diese Werte werden anschließend mittels Volumerendering in eine 3D-Szene überführt.

### 3.1. Mathematische Beschreibung von Szenen

NeRFs repräsentieren eine statische Szene als kontinuierliche 5D-Funktion  $f : (x, y, z, \theta, \phi) \rightarrow (r, g, b, \sigma)$ . Dabei werden alle Punkte der Szene  $(x, y, z)$  mit Orientierung  $\theta$  und

Richtung  $\phi$  abgebildet auf einen abgestrahlte Farbe  $c = (r, g, b)$  und einen Dichtewert  $\sigma$ .  $\sigma$  fungiert dabei als differenzielle Trübung, die kontrolliert, wie viel Strahlungsintensität akkumuliert wird durch eine Strahl  $r(t) = o + td$ , der den Punkt  $(x, y, z)$  passiert. Diese Funktion wird mit Hilfe eines voll-verbundenen neuronalen Netzwerkes ohne Faltung, sogenannte MLP,  $F_\Theta : (x, d) \rightarrow (c, \sigma)$  approximiert. Das Ergebniss dieses MLPs ist eine koordinatenabhängiger Dichtewert, sowie eine koordinaten- und richtungsabhängige, abgestrahlte Farbe. Um diese Richtungsabhängigkeit der Farbe zu realisieren, verarbeitete das Netzwerks zuerst den Input der 3D-Koordinate un produziert den Dichtewert  $\sigma$ , welcher dann zusammen mit den Richtungsinfoamtionen erneut an ein Netzwerk übergeben wird, um einen richtungsabhängigen Farbwert zu generieren.

### 3.2. Volume Rendering

Rein technisch geschieht das Rendering der Szene [8] über virtuelle Lichtstrahlen, welche von virtuellen Kameras durch die Szene geschossen werden. Die Farbe jedes Strahls wird dabei mit Hilfe von Volumen Rendering bestimmt. Der Dichtewert  $\sigma(x)$  kann dabie als differentielle Wahrscheinlichkeit gesehen werden, mit der ein Strahl an eim infinitesimalen Partikel an der Position  $x$  terminiert. Die erwartete Farbe  $C(r)$  eines Kamerastrahls  $r(t) = o + td$  mit Nah- und Ferngrenzen  $t_n, t_f$  ist:

$$C(r) = \int_{t_n}^{t_f} T(t)\sigma(r(t))c(r(t), d)dt \quad (1)$$

wobei

$$T(t) = \exp\left(-\int_{t_n}^t \sigma(r(s))ds\right) \quad (2)$$

$T(t)$  beschreibt dabei die akkumulierte Strahlkraft entlang eines Strahls von  $t_n$  zu  $t$ , wenn keine Partikel getroffen werden. Um eine Szene des NeRFs zu rednern, muss  $C(r)$  für je einen Strahl pro Pixel der entsprechenden virtuellen Kamera berechnet werden. Das Integral wird mittels numerischer Quadratur approximiert. Um nicht die Auflösung des MLPs einzuschränken, wird statt deterministischer Approximation ein statistischer Ansatz verwendet, bei dem das Intervall  $[t_n, t_f]$  in  $N$  gleichverteilte Abschnitte aufgeteilt wird und dann jede Probe mit gleicher Wahrscheinlichkeit aus einem der Abschnitte gezogen wird, sodass:

$$t_i \sim U\left[t_n + \frac{i-1}{N}(t_f - t_n), t_n + \frac{i}{N}(t_f - t_n)\right] \quad (3)$$

Damit kann nun  $C(r)$  approximiert werden zu:

$$\hat{C}(r) = \sum_{i=1}^N T_i(1 - \exp(-\sigma_i \delta_i))c_i \quad (4)$$

wobei

$$T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right) \quad (5)$$

$\delta_i = t_{i+1} - t_i$  ist dabei der Abstand zwischen zwei Proben.

### 3.3. Multiresolution Hashgrid encoding

Instant-NGP löst dabei das Hauptptoblem von Szenengenerierung mit NeRFs: die langen Trainingszeit des MLP auf Grund ineffizienter Kodierung der Pixel- Positionsweite. Für eine

gegebenes voll-verbundenes neuronales Netzwerk  $m(y; \phi)$  bietet Multiresolution Hashgrid encoding eine Kodierung der Inputs  $y = enc(x, \theta)$ , welche die Qualität und Geschwindigkeit der Approximation von  $f : (x, y, z, \theta, \phi) \rightarrow (r, g, b, \sigma)$  erhöht, ohne den Ressourcenverbrauch nennenwert zu erhöhen.

Dazu wird das bestehende MLP mit trainierbaren Gewichten  $\phi$  um trainierbare kodierungsparameter  $\theta$  erweitert. Diese Parameter werden in  $L$  Level aufgeteilt, wobei jedes Level bis zu  $T$  Featurevektoren (Position, Orientierung der Pixel) von Dimension  $F$  beinhaltet. Dabei ist jedes Level unabhängig und speichert die Features an den Eckpunkten eines Rechtecks, wobei die Auflösung des Rechtecks eine geometrische Sequenz zwischen der gröbsten und der feinsten Auflösung  $[N_{min}, N_{max}]$  ist:

$$N_l := \lfloor N_{min} * b^l \rfloor, \quad (6)$$

$$b := \exp\left(\frac{\ln N_{max} - \ln N_{min}}{L - 1}\right) \quad (7)$$

Betrachten wir ein einzelnes Level  $l \in L$ . Die Inputkoordinate  $x \in \mathbb{R}^d$  wird auf die Auflösung von  $l$  skaliert, bevor sie auf- und abgerundet wird, sodass  $\lfloor x_l \rfloor := \lfloor x^* N_l \rfloor$ ,  $\lceil x_l \rceil := \lceil x^* N_l \rceil$ .  $\lfloor x_l \rfloor$  und  $\lceil x_l \rceil$  spannen ein Gitter mir  $2^d$  Intergerwerten aus  $\mathbb{Z}^d$  auf. Jede Ecke des Gitters wird abgebildet auf einen Eintrag in der Hashmap des jeweiligen Levels  $l$ . Für grobe Auflösungen geschieht diese Abbildung  $1 : 1$ , für feinere Level wird eine Hashfunktion  $h : \mathbb{Z}^d \rightarrow \mathbb{Z}_T$  ohne Kollisionsbehandlung verwendet. Stattdessen werden eventuelle Kollisionen und dadurch entstehender Detailverlust durch eine hohe Präzision der Ergebnisse des MLPs wieder ausgeglichen. Die Hasfunktion hat dabei die Form

$$h(x) = (\oplus_{i=1}^d x_i \pi_i) \bmod T. \quad (8)$$

Dabei stellt  $\oplus$  den Bitweisen XOR-Operator dar und  $\pi_i$  sind universelle Primzahlen. Zuletzt werden die Featurevektoren an der Ecken d-linear interpoliert auf die relative Position des Inputs  $x$ , mit Hilfe der Interpolationsparameter  $w_l := x_l - \lfloor x_l \rfloor$ .

Instant-NGP schafft es mit Hilfer des Multiresolution Hasgrid encodings, die Trainigszeit des MLPs dramatisch zu reduzieren. Während erste Implementationen von NeRFs noch Trainingszeiten im Stundenbereich hatten, werden Instant-NeRFs innerhalb weniger Minuten mit vergleichbarer Qualität produziert.

### 3.4. Nutzung von Instant-NGP

Instant-NGP muss auf Linux basierten Systemen neu kompiliert werden. Ich bin dafür auf meinem System wie folgt vorgegangen:

```
sudo apt-get install \
    build-essential \
    git \
    python3-dev \
    python3-pip \
    libopenexr-dev \
    libxi-dev \
    libglfw3-dev \
    libglew-dev \
    libomp-dev \
    libxinerama-dev \
    libxcursor-dev

export PATH="/usr/local/cuda-<your_cuda_version>/bin:$PATH"
```

```

export LD_LIBRARY_PATH="/usr/local/cuda-<your_cuda_version>/
lib64:$LD_LIBRARY_PATH"

$ git clone --recursive https://github.com/nvlabs/instant-ngp
$ cd instant-ngp

$ cmake . -B build -DCMAKE_BUILD_TYPE=RelWithDebInfo
-DCMAKE_CUDA_COMPILER=/usr/local/cuda-<your_cuda_verision>/bin/nvcc
$ cmake --build build --config RelWithDebInfo -j

```

Die Spezifizierung der benutzen CUDA-Version ist dabei nur notwendig, falls auf dem benutzen System so wie in meinem Fall mehrere CUDA-Versionen installiert sind.

Instant-NGP benötigt für die Rekonstruktion von Szenen zwei Arten von Inputdaten:

- Geeignete Bilddaten
- Kameraposen

Im Bezug auf die geeigneten Bilddaten werde ich in einem späteren Abschnitt noch einmal zu sprechen kommen. Die Entwickler von Instant-NGP empfehlen explizit, COLMAP zur Generierung von Kameraposen zu nutzen. Dazu wird von NVIDIA ein Python-Skript zur Verfügung gestellt, welches die Umwandlung des COLMAP-Outputs in ein für Instant-NGP nutzbares Format übernimmt.

### **3.5. Anforderungen an Bilder für NeRF-Generierung**

Bis zum Schluss meiner Arbeit mit Instant-NGP habe ich keine finalen Weg gefunden, konsequent Datensets von Szenen zu generieren, welche beim Training zufriedenstellende Resultate erzielten. Durch ausgedehnte Tests habe ich jedoch einige Faktoren herausarbeiten können, welche die Qualität der produzierten Szenen maßgeblich beeinflussen. Diese Liste ist höchstwahrscheinlich nicht final, umfasst aber mein aktuelles Verständnis an Einflussfaktoren.

#### **3.5.1. Aufnahmebereich**

Je mehr der Szene pro Bild abgebildet wird, desto besser wird die anschließende Rekonstruktion. Demnach empfiehlt es sich, Kameras mit weiten Sichtbereichen, z.B. Fischaugenlinsen, zu verwenden. In den gleichen Bereich fällt auch, dass zwischen den einzelnen Bildern des Datensatzes eine gewisse Überlappung vorhanden sein muss. Andernfalls entstehen bei der Generierung der Kameraposen durch COLMAP und subsequent bei der Rekonstruktion große Fehler in der Szene. Dieser Punkt wird insbesondere relevant, wenn die Bilder des Datensatzes aus Videoaufnahmen der Szene gewonnen werden sollen. In diesem Fall ist es kritisch, alle Bereiche der Szene möglichst gleichmäßig abzubilden, um fehlerhafte Regionen im NeRF zu vermeiden.

Die Vorgabe von NVIDIA für die optimale Anzahl an Bildern im Trainingsdatensatz liegt bei etwa 50-150 Bildern. Generell lässt sich sage, das mehr Bilder im Datensatz auch zu einer besseren Rekonstruktion der Szene führen. Allerdings ist die ideale Anzahl an Bildern auch nach oben hin beschränkt. Dies hat zwei Gründe: Zum einen steigt mit steigender Anzahl an Bildern die Laufzeit von COLMAP sehr stark an. Zum anderen scheinen aber auch bei Mehrfachabbildung mancher Bereiche der Szenen, die Fehler in diesem Bereich wieder zu steigen, eventuell auf Grund von Überanpassung des NeRFs.

#### **3.5.2. Auflösung der Kamera**

Nach den Anforderungen an einem möglichst großen Abbildungsbereich der Kamera stellt die Auflösung der verwendeten Kamera vielleicht die wichtigste Anforderung dar. Während meiner Arbeit mit Instant-NeRF hat sich sehr schnell herausgestellt, dass eine höhere Auflösung der

Trainingsbilder auch zu einer höheren Qualität der NeRFs führt. Beispiele für diese Erkenntnis werden im Ergebnisabschnitt diskutiert.

### 3.5.3. Gleichmäßigkeit des Datensatzes

Die Bilder des Trainingdatensatzes sollten in einigen Punkten gleichmäßig sein:

- Schärfe: Je schärfer die Bilder, desto besser die Rekonstruktion. Hinzu kommt, dass sich die Bilder untereinander nicht sonderlich in ihrer Schärfe unterscheiden sollten. Demnach sollte vor allem Bewegungsunschärfe und schlechte Fokussierung vermieden werden. Da insbesondere das Multiresolution Hashgrid Encoding sich nach der Auflösung von Details in den individuellen Bildern richtet ( $N_{max}$ ), können einige schlecht aufgelöste oder verwackelte Bilder im Datensatz zu einer allgemein geringeren Detailauflösung führen, sodass auch Details von gut aufgelösten Bildern nicht mehr reproduziert werden können
- Helligkeit: Ähnlich wie auch bei der Schärfe sollten die Bilder des Datensatzes ähnliche Helligkeitswerte aufweisen, um Fehler während des Trainings zu vermeiden.
- Position/Rotation der Szene: Instant-NGP ermöglicht eine erfolgreiche Rekonstruktion **statischer** Szenen. Ändern sich innerhalb des Trainingsdatensatzes die Position oder Rotation der Szene, wird die Szene also dynamisch, so ist keine Rekonstruktion mehr möglich.

## 3.6. Ergebnisse

Die grundlegende Überlegung beim Einsatz von Instant-NGP war es, den bereits bestehenden Datensatz des Fraunhofer IPK aus dem EIBA/MVIP Projekt zu nutzen, um Rekonstruktionen der in den Szenen abgebildeten Objekten zu erhalten. Allerdings musste ich schnell feststellen, dass sich dieser Datensatz nicht zur NeRF-Generierung eignete, da die Bilder des Datensatzes jeden Punkt der oben genannten verfehlten. Die einzelnen Bilder wurden mittels einer Intel RealSense Kamera aufgenommen, welche mit einer Auflösung von 1920x1080p vergleichsweise schlecht aufgelöste Bilder produzierte:



Figure 3: Beispielbilder aus EIBA/MVIP Trainingsdatensatz

Das größte Problem war jedoch, dass auf Grund der Konstruktion des Datensatzes zwischen einzelnen Bildern die Orientierung des Objektes geändert wurde, sodass Instant-NGP keine nutzbare Rekonstruktion lieferte.

### 3.6.1. Intel RealSense

Somit produzierte ich eigene Trainingsdaten, zunächst mit der selben Kamera, mit der auch der MVIP Datensatz produziert worden war. Um die Datengenerierung auch mit Blick auf eine mögliche Skalierung des Projekts möglichst einfach zu gestalten, entschied ich mich für eine videobasierte Aufnahme der Szenen. Mittels Python-Skript wurden dann aus den Videoaufnahmen einzelne Frames generiert, sodass am Schluss ein Trainingsdatensatz von ungefähr 50-150 Bildern existierte. Das Hauptproblem war, dass sowohl die Szenenqualität gering war und die aus den Szenen produzierten Meshes sehr schlecht aufgelöst waren, hier am Beispiel von Objekt EIBA-5-2:



Figure 4: Mesh und Instant-NGP Szene für RealSense Datensatz

### 3.6.2. Microsoft Azure Kinect DK

Um die Qualität des NeRFs und des erstellten Meshes weiter zu verbessern, wechselte ich zu einer Kamera mit höherer Auflösung, die Microsoft Azure Kinect DK mit einer Auflösung von 3920x2160p. Auch erlaubte es die Azure Kinect DK, Videos mit höherer Framerate (30fps) als die Intel RealSense (15fps) zu tätigen. Ich erhoffte mir davon eine bessere Gleichmäßigkeit in dem erstellten Trainingsdatensatz zu erhalten. Außerdem erweiterte ich die Vorverarbeitung der Daten um zwei Algorithmen, welche Bilder mit starken Abweichungen vom Mittel des Datensatzes im Bezug auf Helligkeit und Schärfe aussortierten. Diese Anpassungen der Verarbeitung und der Hardware sorgten weder für eine Steigerung der Qualität der NeRFs, noch der Meshes, eher gegenteilig:



Figure 5: Mesh für AzureKinect DK Datensatz

### 3.6.3. Apple Iphone 11

Nach erneuter Analyse der von NVIDIA bereitgestellten Vergleichsdaten entschied ich mich dazu, weg von videobasierter Datengenerierung hin zu fotobasierte Generierung zu gehen. Dazu wechselte ich das Aufnahmegerät zu Kamera des Iphones 11. Zusätzlich wählte ich nun selbst aus, aus welcher Perspektive die Szene fotografiert wurde. Der damit erstellt Datensatz umfasste nun nur noch ca. 50 Bilder, allerdings mit einer Auflösung von 4032x3024p. Außerdem konnte ich durch manuelle Aufnahmen der Bilder für jedes Bild die oben genannten Anforderungen sicherstellen:



Figure 6: Beispielbilder aus Iphone 11 Trainingsdatensatz

Nicht nur führte diese Änderung zu einer dramatischen Verbesserung der NeRFs, auch konnte ich aus diesen NeRFs Meshes mit hoher Detailsauflösung extrahieren:



Figure 7: Mesh für Ipone11 Datensatz

Dennoch muss ich abschließend festhalten, dass Instant-NeRF nicht für die gewollte Aufgabe brauchbar ist. Zwar weisen die generierten NeRFs eine hohe Detailauflösung im Vergleich zu den Referenzbildern auf, doch bleibt die Qualität der generierten Meshes hinter den Ergebnissen des NeRFs zurück. Dies liegt daran, dass Instant-NGP nicht für die Generierung von Meshes aus den erstellten Szenen ausgelegt ist. Der integrierte Meshing-Algorithmus ist nur sehr rudimentär implementiert das NeRF generiert weder eine Textur noch ein Material für die rekonstruierte Szene. Dieser Umstand macht Instant-NGP für meine Aufgabe der Erstellung synthetischer Daten leider unbrauchbar.

## 4. Nvdiffrc

Nvdiffrc [9] arbeitet bei der Rekonstruktion von Szenen mit einem anderen Ansatz als Instant-NGP. Im Gegensatz zu Instant-NGP ist nvdiffrc tatsächlich auf die Rekonstruktion einzelner Objekte ausgelegt und nicht auf die Rekonstruktion ganzer Szenen. Somit generiert nvdiffrc nicht nur ein Mesh des Objekts, sondern dazu noch eine Textur, ein Material und approximiert außerdem einen Renderer, der die Lichtverhältnisse der Referenzbilder optimal abbildet.

### 4.1. Mathematische Beschreibung von Szenen

Nvdiffrc repräsentiert Szenen nicht mehr als NeRF, sondern wählt den Ansatz über ein Deep Marching Tetrahedra Mesh (DMTet) [10] in Kombination mit einem Signed Distance Field (SDF). Damit vereint nvdiffrc zwei unterschiedliche Varianten in der Repräsentation dreidimensionaler Szenen und erweitert dazu noch DMTet, auch auf zweidimensionale Daten trainierbar zu sein.

SDFs sind eine implizite Beschreibungsform dreidimensionaler Objekte. Ein SDF ist dabei im Prinzip nur eine Sammlung von Funktionen, welche für einen gegebenen Punkt  $p \in \mathbb{R}^3$  eine Aussage darüber treffen, ob der Punkt sich in dem Objekt befindet oder nicht. Nimmt das SDF den Wert 0 an, so befindet sich der Punkt genau auf der Kante des Objekts, ein Wert kleiner 0 bedeutet, der Punkt

befindet sich innerhalb des Objektes, ein Wert größer 0 bedeutet, dass der Punkt außerhalb des Objektes liegt. SDFs bilden damit eine sehr effiziente Methode, beliebige Geometrien abzubilden und Kollisionen mit anderen Punkten oder Geometrien zu prüfen.

Nvdiffrec und DMTet repräsentieren Meshes als Tetraeder-Raster ( $V_T, T$ ). wobei  $V_T$  alle Punkte des Rasters  $T$  sind. Jedes Tetraeder  $T_k \in T$  wird repräsentiert über vier Eckpunkte  $\{v_{a_k}, v_{b_k}, v_{c_k}, v_{d_k}\}$ , mit  $k \in \{1, \dots, K\}$ , wobei  $K$  die Gesamtzahl an Tetraedern des Rasters ist und  $v_{i_k} \in V_T$ .

Nvdiffrec erweitert die Idee des SDF als Neuronales SDF, wobei die Parameter des SDF nun als trainierbare Gewichte eines neuronalen Netzwerkes abgebildet werden, erweitert um Parameter für die Beschreibung eines räumlich variierenden Materials und für die Belichtung der Szene. Sei  $\Phi$  die Menge dieser Parameter. Für eine gegebene Kamerapose  $c$  produziert der differentielle Renderer von nvdiffrer ein Bild  $I_\Phi(c)$ , mit korrespondierendem Referenzbild  $I_{\text{ref}}(c)$ . Das neuronale SDF optimiert nun während des Trainings für einen gegebenen Verlustfunktion  $L$  den Ausdruck

$$\arg \min_{\Phi} \mathbb{E}_c [L(I_\Phi(c), I_{\text{ref}}(c))] \quad (9)$$

Der Renderer von nvdiffrer produziert das Referenzbild anhand eines während des Renderings produzierten Meshes. Dazu wandelt nvdiffrer die implizite Geometriedarstellung des Objektes bestehend aus Tetraeder-Raster und SDF in eine explizite Darstellung in Form eines Meshes um.

Damit adaptiert nvdiffrer den Ansatz von DMTet, mit dem Meshes anhand von dreidimensionalen geometrien trainiert werden. Durch die Integration von Referenzbildern und dem Vergleich mit gerenderten Bildern, schafft nvdiffrer es, die Verbesserung von Oberflächenstrukturen, die DMTet implementiert, mit dem allgemeinen Problem der Rekonstruktion anhand von Bilddaten zu verbinden.

## 4.2. Rendering

Neben einer anderen Darstellung der Szenen verwendet nvdiffrer auch einen anderen Ansatz für das Rendern der einzelnen Szenen als Instant-NGP. Anstelle des von Instant-NGP verwendeten Volume Renderings, welches für die wiederholte Anwendung während des Trainings ineffizient wäre und keine optimalen Ergebnisse liefert, verwendet nvdiffrer drei verschiedene Komponenten innerhalb des Renderers für Textur, Material und Belichtung. Für die Generierung des Materials verwenden die Entwickler einen von Disney [11] entwickelten Ansatz, der Rekonstruktion von Materialien anhand von physikalischen Vergleichsmaterialien optimiert. Für die Textur adaptiert nvdiffrer die Idee des MLP, also eines neuronalen Netzwerks, welches, ähnlich wie bei Instant-NGP, anhand einer gegebenen Position eine positionskodierte Textur ermittelt.

Da an einer anderen Stelle dieser Dokumentation erneut auf die von nvdiffrer verwendete Renderingmethode eingegangen wird, möchte ich diese hier etwas genauer betrachten. Um hier nicht komplette Ergebnisse aktueller Forschung abbilden zu müssen und um die Komplexität zu reduzieren, hier ein kurzer Überblick:

Nvdiffrer wählt für die Repäsentation von Szenebelichtung einen Ansatz über ausgehende Strahlkraft. Diese richtungsabhängige Strahlkraft  $L(\omega_0)$  in Richtung  $\omega_0$  kann beschrieben werden über:

$$L(\omega_0) = \int_{\Omega} L_i(\omega_i) f(\omega_i, \omega_0) (\omega_i * n) d\omega_i. \quad (10)$$

Dabei ist  $L_i(\omega_i)$  die eingehende Strahlkraft aus Richtung  $\omega_i$  und  $f(\omega_i, \omega_0)$  eine Funktionsklasse zur Beschreibung von Reflexion an nicht idealen Oberflächen ist. In diesem Fall wird ein Modell für

Reflektion an rauen Oberflächen verwendet [12]. Sei im weiteren  $D$  eine Modellierung eben dieser Oberflächen. Die Approximation von (10) geschieht wie folgt:

$$L(\omega_0) = \int_{\Omega} f(\omega_i, \omega_0)(\omega_i * n) \int_{\Omega} L_i(\omega_i) D(\omega_i, \omega_0)(\omega_i * n) d\omega_i. \quad (11)$$

Das Problem bei dieser Art von Lichtmodellierung ist ein hoher Leistungsoverhead während der Laufzeit. Eine Lösungsmöglichkeit stellt eine nicht-deterministische Integration dar, sogenannte Monte-Carlo Integration, welche aber aufgrund hoher Fehleranfälligkeit von den Autoren verworfen wird. Diese Fehleranfälligkeit ist auf eine hohe empirische Varianz des statistischen Integrationsansatzes zurückzuführen, was zu starkem Bildrauschen beim Rendern führt. Da nvdiffrast das zu Grunde liegende Netzwerk anhand von gerenderten Referenzbildern optimiert, bedeutet starkes Bildrauschen in den Referenzbildern zu einem schlechteren Gesamtergebnis.

### 4.3. Nutzung von nvdiffrast

Nvdiffrast wird in eine eingens konfigurierte Conda Umgebung eingebunden. Da nvdiffrast auf einem Pytorch Netzwerk aufbaut, muss bei der Konfiguration insbesondere auf die richtige Konfiguration von PyTorch geachtet werden:

```
conda create -n dmodel python=3.9
activate dmodel
export LD_LIBRARY_PATH="/usr/local/cuda-<your_cuda_version>/lib64:$LD_LIBRARY_PATH"
conda install conda install -c "nvidia/label/cuda-<your_cuda_version>" cuda-toolkit
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/
whl/cu<your_cuda_version>
pip install ninja imageio PyOpenGL glfw xatlas gdown opencv-python
pip install git+https://github.com/NVlabs/nvdiffrast/
pip install --global-option="--no-networks" git+https://github.com/NVlabs/tiny-cuda-
nn#subdirectory=bindings/torch
```

*opencv* ist laut den Entwicklern keine explizite Voraussetzung. Allerdings habe ich auf meinem System feststellen müssen, dass ohne *opencv* während der Laufzeit von nvdiffrast kein Export von Checkpoints möglich ist, wodurch keine Kontrolle der Ergebnisse während der Laufzeit stattfinden kann. Außerdem schlägt der finale Export der Shadinginformationen fehl, da die von nvdiffrast verwendet Bibliothek *imageio* beim Export von Dateien ein *opencv* backend benutzt.

Nvdiffrast arbeitet bei der Rekonstruktion von Szenen mit einem anderen Ansatz als Instant-NGP. Im Gegensatz zu Instant-NGP ist nvdiffrast tatsächlich auf die Rekonstruktion einzelner Objekte ausgelegt und nicht auf die Rekonstruktion ganzer Szenen. Somit generiert nvdiffrast nicht nur ein Mesh des Objekts, sondern dazu noch eine Textur, ein Material und approximiert außerdem einen Renderer, der die Lichtverhältnisse der Referenzbilder optimal abbildet.

### 4.4. Anforderungen an Trainingsdatensatz

Nvdiffrast stellt für eine erfolgreiche Rekonstruktion andere Anforderungen an den Trainingsdatensatz als Instant-NGP. Einige Anforderungen bleiben erhalten, so etwa die Anforderungen nach gleichbleibender Belichtung und Schärfe der einzelnen Bilder. Die Forderung nach einer statischen Szene wird allerdings etwas aufgeweicht. So ist es in der Erstellung von Trainingsdaten für nvdiffrast optimal, wenn eine komplette Abdeckung des Objektes durch die Bilder durch eine Rotation des Objektes geschieht, im Gegensatz zu einer Änderung der Kameraposition im Falle von Instant-NGP. Der Unterschied zwischen den Daten lässt sich am besten in einer Visualisierung der Kameras mit COLMAP verdeutlichen:



Figure 8: Kameraposen Trainingsdaten: links Instant-NGP, rechts nvdiffrc

Diese Vorgehensweise vereinfacht die Datenerzeugung massiv. Denn durch eine statische Kamera ist es leichter, konsequent eine gute Abdeckung des Objektes zu gewährleisten, sowie eine gute Qualität der einzelnen Bilder sicherzustellen. Auch ist diese Art der Datenerzeugung skalielierbar, da z.b mit Hilfe einer rotierenden Scheibe die Datenerzeugung nahezu automatisiert werden kann. Ein weiterer Unterschied zu Instant-NGP ist auch die benötigte Anzahl an Bildern. Während Instant-NGP 50-150 Bilder für eine erfolgreiche Rekonstruktion benötigt, reichen von nvdiffrc etwa 40-50 Bilder. Allerdings gilt auch hier, dass unzureichende Bilder im Training auch zu einem schlechten Ergebnis führen werden.

Der größte Unterschied zwischen den beiden Methoden ist jedoch, dass nvdiffrc für die Rekonstruktion Objektmasken benötigt.



Figure 9: nvdiffrc Trainingsdaten

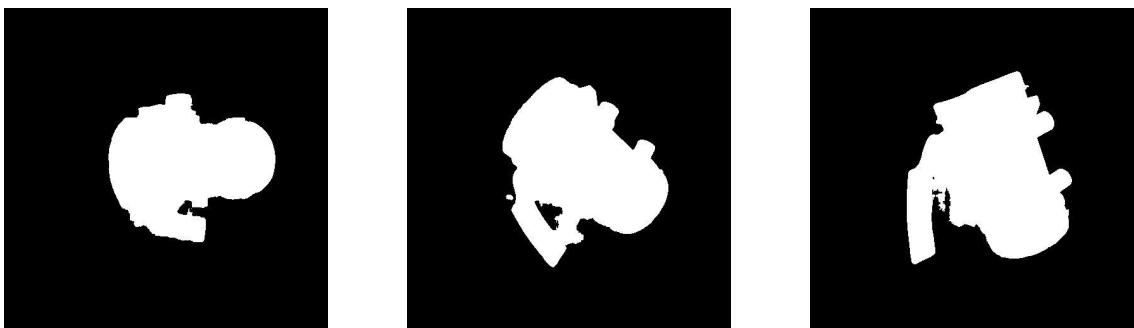


Figure 10: korrespondierende Objektmasken

Die Masken wurden mittels des im Rahmen des Fraunhofer MVIP Projektes entwickelten Segmentationsmodells erstellt.

#### 4.5. Ergebnisse

Die von Nvdiffrc produzierten Ergebnisse sind stark von der Qualität der von COLMAP generierten Kameraposen abhängig. Dementsprechend habe ich am Anfang starke Schwierigkeiten gehabt, mit

den von COLMAP generierten Kameraposen hochwertige nvdiffrec Modelle zu generieren, geschweige denn, die von NVIDIA in ihrem Paper beschriebenen Resultate zu reproduzieren. Eine genaue Beschreibung der Probleme und der Lösungen findet sich in Abschnitt 2.5.

Vor dem Start des Trainings muss für den jeweiligen Datensatz eine Trainingskonfiguration erstellt werden, hier am Beispiel des von NVIDIA bereitgestellte Datensatzes [4]:

```
{
    "ref_mesh": "./workspace/images_rescaled",
    "random_textures": true,
    "isosurface": "flexicubes",
    "iter": 5000,
    "save_interval": 100,
    "texture_res": [2048,2048],
    "train_res": [1024,1024],
    "batch": 12,
    "learning_rate": [0.03,0.03],
    "kd_min": [0.03,0.03,0.03],
    "kd_max": [0.8,0.8,0.8],
    "ks_min": [0,0.08,0], "ks_max": [0,1.0,1.0],
    "dmtet_grid": 128,
    "mesh_scale": 2.5,
    "camera_space_light": true,
    "background": "white",
    "display": [{"bsdf": "kd"}, {"bsdf": "ks"}, {"bsdf": "normal"}],
    "out_dir": "mod_ehead_largetgrid"
}
```

In dieser Konfiguration sind drei Punkte wesentlich für die Qualität der Reproduktion verantwortlich:

- *train\_res*
- *batch\_size*
- *dmtet\_grid*

*Train\_res* steuert die Auflösung der Bilder, die zur Rekonstruktion verwendet werden. Eine höhere Auflösung bedeutet eine höhere Detailauflösung in den Bildern, was grundsätzlich zu einer höheren Detailauflösung der Rekonstruktion führt. Allerdings trifft dies nicht immer zu, wie das folgende Beispiel zeigt:



Figure 11: Detailvergleich: links 512x512, rechts 1024x1024 in Reihenfolge: Referenz, Mesh+Textur, Rendering

Beide Modelle wurden mit der gleichen *batch\_size* von 12, sowie einer Auflösung des zu Grunde liegenden DMTet-Rasters von 128 trainiert. Zu sehen ist, dass bei der Rekonstruktion mit der höheren Auflösung der Trainingsbilder die Details des Labels des Objektes nicht so gut dargestellt werden, wie beim Vergleich mit dem Trainingsdatensatz mit geringerer Auflösung. Über den Grund dieser negativen Korrelation zwischen Trainingsauflösung und Detailauflösung der Rekonstruktion kann ich nur Spekulationen anstellen. Eine mögliche Erklärung könnten ungenaue COLMAP Kameraposen im Falle der 1024x1024 Auflösung sein, auch wenn sich diese Vermutung auf Grund sehr kleiner Abweichungen nicht abschließend klären lässt.

In der mit nvdiffrcc mitgelieferten Dokumentation [9] findet sich keine Erklärung von *batch\_size*. Vermutlich regelt *batch\_size* die Größe der pro Iteration gezogenen Proben für Textur und Belichtungsapproximation. Dies würde auch erklären, warum *batch\_size* von allen Trainingsparametern mit Abstand den stärksten Einfluss auf den Speicherverbrauch des Trainings hat. Soweit ich das durch meine Tests beurteilen kann, ist der Grafikspeicherverbrauch von nvdiffrcc ungefähr linear abhängig von *batch\_size*. Mit steigender *batch\_size* steigt allerdings nicht zwangsläufig die Qualität der Rekonstruktion.



Figure 12: Vergleich von Rekonstruktionen mit batchsizes von 4, 8, 12 und 24 (links nach rechts)

Alle vier Modelle wurden mit der gleichen Auflösung von 512x512 erstellt. Offensichtlich unterscheidet sich die Qualität der einzelnen Modell nur sehr gering.

*Dmtet\_grid* regelt die Auflösung des DMTes-Rasters für die Repräsentation der Szene, siehe Abschnitt, 4.2 . Eine höhrere Auflösung des DMTet-Rasters sorgt für glattere Oberflächen in der Rekonstruktion, kann jedoch die Detailauflösung der Rekonstruktion reduzieren.

Ein wesentlicher Teil der Arbeit mit nvdiffrcc war, herauszufinden, welche Trainingskonfiguration die best möglichen Ergebnisse lieferte. Obwohl die Arbeit an nvdiffrcc auf einem sehr leistungsstarken System ablief, stieß ich doch bei der Festlegung der Trainingsparameter an gewisse Grenzen. Wie bereits beschrieben, ist der Speicherverbrauch des Trainings von nvdiffrcc linear abhängig von *batch\_size*. Für Trainingsauflösungen von 512x512 Pixel sind Werte von *batch\_size*  $b \in [4, 32]$  plausibel, ohne den zu Verfügung stehenden Grafikspeicher komplett auszunutzen, oder einen starken Abfall der Qualität zu generieren. Wie oben bereits gezeigt, führt eine erhöhung der *batch\_size* nicht zwangsläufig zu einer besseren Qualität der Rekonstruktion. Für eine Trainingsauflösung von 1024x1024 Pixel reduziert sich dieses Intervall zu  $b \in [4, 12]$ , wobei ein Training mit Auflösung 1024x1024 und *batch\_size* von 12 die 49GB verfügbaren Grafikspeicher fast vollständig auslasten.

Die besten Ergebnisse erziehlt nvdiffrcc interressanterweise an den komplexesten Objekten. Das

liegt wahrscheinlich daran, dass für komplexe Geometrien die von COLMAP generierten Kameraposen am präzisesten sind, was sich wiederum positiv auf die Qualität der Rekonstruktion auswirkt.



Figure 13: Detailvergleich: Referenz, Textur+Mesh, Rendering

Für unkomplexe Geometrien und Objekte und Objekte mit stark reflektierenden Oberflächen hat COLMAP Schwierigkeiten präzise Posen zu generieren, oder die Posengenerierung schlägt grundlegend fehl. Auch gestaltet sich die Maskengenerierung für Objekte dieser Art schwierig, was zu einer allgemein schlechten Rekonstruktion führt:



Figure 14:

Fehlerhafte Referenzbilder durch ungenaue Maskengenerierung (oben) mit Fortpflanzung in Renderings (unten)

#### 4.5.1. Laufzeit

Nvdiffrc hat im Vergleich zu Instant-NGP deutlich längere Laufzeiten. So trainiert Instant-NGP ein NeRF zur Repräsentation einer Szene innerhalb weniger Minuten, während die Trainingszeiten von nvdiffrc im Bereich von ein paar Stunden, abhängig von der genutzten Konfiguration, liegen. Die um beide Methoden entwickelte Pipeline hat in beiden Fällen ähnliche Laufzeiten, wobei bei Instant-NGP auf Grund der größeren Anzahl an Trainingsbildern insbesondere COLMAP den meisten Overhead verursacht, während bei nvdiffrc die Maskengenerierung die meiste Zeit in Anspruch nimmt.

#### 4.5.2. Beste Ergebnisse und Konfiguration

Die mit Abstand besten Ergebnisse konnte ich mit folgender Konfiguration erzielen:

```
{
    "ref_mesh": "workspace/images_rescaled",
    "random_textures": true,
    "isosurface": "flexicubes",
    "iter": 5000,
    "save_interval": 100,
    "texture_res": [2048,2048],
    "train_res": [1024,1024],
    "batch": 4,
    "learning_rate": [0.03,0.03],
    "kd_min": [0.03,0.03,0.03],
    "kd_max": [0.8,0.8,0.8],
    "ks_min": [0,0.08,0],
    "ks_max": [0,1.0,1.0],
    "dmtet_grid": 128,
    "mesh_scale": 5.0,
    "camera_space_light": true,
    "background": "white",
    "display": [{"bsdf": "kd"}, {"bsdf": "ks"}, {"bsdf": "normal"}]
},
    "out_dir": "foo"
}
```

Die Nachfolgenden Beispiele haben immer das gleiche Format in Reihenfolge: Rendering, Referenz, Mesh, Oberflächenrenering, Normalenrendering:



Figure 15:

## 5. Nvdiffrecmc

Nvdiffrecmc[13] baut auf nvdiffrec als Methode auf, verbessert allerdings die Performance von nvdiffrec an zwei Punkten. Einmal stellen die Autoren eine Methode zur Varianzreduktion der Monte-Carlo Integration vor. Zum anderen verwendet nvdiffrcmc eine neuronale Rauschunterdrückung, um das Bildrauschen der für die Optimierung der Methode gerenderten

Bilder weiter zu reduzieren. Zm Zeitpunkt des Abschlusses dieser Dokumentation habe ich noch keine vorzeigbaren Daten mit nvdiffrecmc generiert.

## 5.1. Monte-Carlo Integration

Monte-Carlo Integration ist ein nicht-deterministischer Ansatz der approximativen Integration. Nicht-deterministisch bedeutet in diesem Fall, dass für unterschiedliche Durchführungen unterschiedlichen Ergebnisse der Approximation zu erwarten sind. Die Monte-Carlo Integration löst dabei das Problem der Approximation eines mehrdimensionalen Integrals

$$I = \int_{\Omega} f(\vec{x}) d\vec{x} \quad (12)$$

wobei  $\Omega \subset \mathbb{R}^m$  eine Teilmenge des m-dimensionalen Raumes mit Volumen

$$V = \int_{\Omega} d\vec{x} \quad (13)$$

ist. Die Monte-Carlo Integration approximiert nun (12), indem  $N$  gleichverteilte Proben  $\vec{x}_1, \dots, \vec{x}_N \in \Omega$  gezogen werden. Damit folgt:

$$I \approx Q_N := V \frac{1}{N} \sum_{i=1}^N f(\vec{x}_i). \quad (14)$$

Wegen des Gesetzes der großen Zahlen gilt:  $\lim_{N \rightarrow \infty} Q_N = I$ .  $Q_N$  stellt dabei eine Verteilungsfunktion der Approximation dar, mit

$$\text{Var}(Q_N) = V^2 \frac{\sigma_N^2}{N}, \quad (15)$$

mit

$$\sigma_N^2 = \frac{1}{N-1} \sum_{i=1}^N (f(\vec{x}_i) - f)^2. \quad (16)$$

Offensichtlich gilt:  $\lim_{N \rightarrow \infty} \text{Var}(Q_N) = 0$ .

Für kleine Stichprobengrößen kann die Varianz jedoch sehr groß werden, was im Bezug zu Methoden wie nvdiffrec zu einem Problem wird, da eine große Varianz, wie schon besprochen, zu starken Bildrauschen führt. Große Stichprobengrößen führen während der Laufzeit jedoch zu lange Berechnungszeiten, sodass Monte-Carlo Integration zunächst für nvdiffrec von den Entwicklern ausgeschlossen wurde. Um diesen Effekt entgegen zu wirken, verwendet nvdiffrecmc den Ansatz des Importance Samplings. Hierbei wird angenommen, dass die Stichprobe  $\vec{x}_1, \dots, \vec{x}_N$  nicht gleichverteilt gezogen wird, sondern aus einer beliebigen Verteilung  $p(\vec{x})$ . Die Idee ist nun, dass  $p(\vec{x})$  so gewählt werden kann, dass die Varianz der Approximation von (12) verringert wird.

## 5.2. Open Image Denoiser

Nvdiffrecmc verwendet für Rauschreduktion ein vortrainiertes Modell [14]. Dieses Modell reduziert das Rauschen der während der Laufzeit produzierten Referenzbilder, sodass der Ansatz zur Optimierung im Gegensatz zu nvdiffrec leicht verändert wird. Sei hier  $D_\theta$  eine Beschreibung der Rauschreduzierung mit Parametern  $\theta$ ,  $I_\Phi(c)$  das während der Laufzeit generierte Bild und  $I_{\text{ref}}(c)$  wieder das Referenzbild, aufgenommen aus der gleichen Kamerapose  $c$ . Nvdiffrecmc optimiert dann den Ausdruck:

$$\arg \min_{\Theta, \Phi} \mathbb{E}_{\epsilon}[L(D_{\Theta}(I_{\Phi}(c)), I_{\text{ref}}(c))]. \quad (17)$$

Die Verlustfunktion  $L$  entspricht der Verlustfunktion, die von nvdiffrer verwendet wird.

### 5.3. Anforderungen an den Datensatz

Auf Grund der Ähnlichkeit von nvdiffrercmc zu nvdiffrer, siehe Abschnitt 4, bleiben die Anforderungen an die Trainingsdaten gleich.

## 6. Dokumentation Codebasis

Sämtliche Software für dieses Projekt wurde in Python implementiert und ist (hier) zu finden. Im Folgenden lege ich eine genauere Beschreibung der einzelnen Funktionalitäten dar, gegliedert nach Methoden.

### 6.1. Instant-NGP

Für die Benutzung von Instant-NGP steht eine vollständige Pipeline zur Verfügung, die die Aufbereitung der Daten und das Training des NeRFs übernimmt. Die Pipeline ist aufgeteilt in drei Python skripte *image2nerf.py*, *nerf2mesh.py* und *toolkit.py*

#### 6.1.1. *image2nerf.py*

*Image2nerf.py* implementiert die Generierung der Kameraposen mittels COLMAP und [15]. In diesem Fall erfolgt die Posengenerierung noch mit Hilfe der Conda Umgebung von COLMAP, da ich zum Zeitpunkt der Implementierung noch keine kompilierte COLMAP Version zur Verfügung hatte. Da keine vollständige Rekonstruktion der Szenen notwendig ist, implementiert das Skript nur die für die Posenbestimmung relevanten Methoden von COLMAP.

#### 6.1.2. *nerf2mesh.py*

*Nerf2mesh.py* implementiert die eigentliche Datenpipeline für das Training der NeRFs. Dazu wird die Arbeitsumgebung von Instant-NGP vorbereitet und alle Trainingsbilder geladen

#### 6.1.3. *toolkit.py*

*Toolkit.py* implementiert eine Reihe von Hilfsmethoden für die Erstellung des Trainingdatensatzes. Insbesondere implementiert *toolkit.py* zwei Vorverarbeitungsalgorithmen

- brightness\_filtering
- sharpness\_filtering

Beide Algorithmen verwenden eine Sliding-Window Ansatz, um über den Datensatz eine Reihe von Schärfe- und Belichtungswerten zu sammeln und Bilder, welche nicht innerhalb einer akzeptablen Abweichung dieses moving averages liegen aus dem Datensatz entfernt.

### 6.2. nvdiffrer

Für nvdiffrer habe ich eine voll funktionstüchtige Pipeline implementiert, welche die Vorverarbeitung der Trainingsdatensätze, die Generierung der Masken und das Training des nvdiffrer Netzwerkes übernimmt. Die Pipeline ist aufgeteilt auf zwei Skripte, *data\_pipeline.py* und *pose\_extraction.py*.

#### 6.2.1. *pose\_extraction.py*

*Pose\_extraction.py* übernimmt die Generierung der Kameraposen durch COLMAP, sowie die Umwandlung des von COLMAP erzeugten Ergebnisses in ein für nvdiffrer verarbeitbares Format.

- read\_cameras\_binary
- read\_points3d\_binary
- read\_images\_binary

Diese Funktionen sind für das Einlesen des von COLMAP produzierten Modells zuständig.

- qvec2rotmat

Generiert eine Rotationsmatrix für die Transformation des von COLMAP verwendeten Koordinatensystems in ein mit Instant-NGP und nvdiffrc/nvdiffrmc kompatibles Koordinatensystem.

- run\_colmap

Führt COLMAP mit einer vordefinierten Konfiguration aus. Da nicht der gesamte Rekonstruktionsprozess benötigt wird, werden nur die in Section 2 beschriebenen Schritte ausgeführt.

- gen\_poses

Generiert die von nvdiffrc für die Rekonstruktion benötigte Posenfile. Dafür Transformiert gen\_poses das COLMAP Koordinatensystem in das nvdiffrc/Instant-NGP Koordinatensystem und legt die Kameramatrix zusammen mit den entsprechenden Bildern in einem mit nvdiffrc kompatiblen Format ab. Diese Funktion habe ich nicht vollständig selbst implementiert, sondern sie baut auf einem von den Entwicklern von Instant-NGP bereitgestellten Skript zur Transformation von COLMAP Koordinaten in Instant-NGP Koordinaten auf [15].

### **6.2.2. data\_pipeline.py**

*Data\_pipeline.py* übernimmt den gesamten Prozess der Rekonstruktion durch nvdiffrc, von Posengenerierung über Vorverarbeitung bis zum eigentlichen Training.

- convert\_heic

Alle für das Training von nvdiffrc aufgenommenen Bilder wurden mit einem Iphone 11 aufgenommen. Apple speichert Bilder im .heic Format ab, welches nicht mit den Standert ImageIO Methoden kompatibel ist. Convert\_heic nutzt die linux-interne heif-convert Bibliothek, um das .heic Format in ein .jpg Format umzuwandeln.

- load\_data

Diese Funktion realisiert die eigentliche Datenpipeline. Load\_data präpariert die Arbeitsumgebung für COLMAP und nvdiffrc, indem alle notwendigen Daten geladen und in den richtigen Verzeichnissen abgelegt werden. Zeilen 179 – 186 sorgen für eine Skalierung der Trainingsbilder in das in der Konfiguration festgelegte Format. Zeilen 188 – 212 regeln die Vordergrundsegmentierung der Objekte mit Hilfe des Fraunhofer IPK MVIP Modells.

## Verweise

- [1] J. L. Schönberger, E. Zheng, M. Pollefeys, and J.-M. Frahm, “Pixelwise View Selection for Unstructured Multi-View Stereo”, in *European Conference on Computer Vision (ECCV)*, 2016.
- [2] J. L. Schönberger and J.-M. Frahm, “Structure-from-Motion Revisited”, in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [3] D. G. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints”, *International Journal of Computer Vision*, vol. 60, pp. 90–110, 2004, doi: 10.1023/B:VISI.0000029664.99615.94.
- [4] [Online]. Available: <https://github.com/vork/ethiopianHead>
- [5] [Online]. Available: <https://github.com/brentyi/ceres-solver>
- [6] T. Müller, A. Evans, C. Schied, and A. Keller, “Instant Neural Graphics Primitives with a Multiresolution Hash Encoding”, *ACM Trans. Graph.*, vol. 41, no. 4, pp. 1–15, Jul. 2022, doi: 10.1145/3528223.3530127.
- [7] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, “NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis”, in *ECCV*, 2020.
- [8] B. v. H. James T. Jajiya, “Ray Tracing Volume Densities”, *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 165–174, Jul. 1984, doi: 10.1145/964965.808594.
- [9] J. Munkberg *et al.*, “Extracting Triangular 3D Models, Materials, and Lighting From Images”, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2022, pp. 8280–8290.
- [10] T. Shen, J. Gao, K. Yin, M.-Y. Liu, and S. Fidler, “Deep Marching Tetrahedra: a Hybrid Representation for High-Resolution 3D Shape Synthesis”, in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.
- [11] B. Burley, “Physically Based Shading at Disney”, in *SIGGRAPH Courses*, 2012.
- [12] H. L. K. E. T. Bruce Walter Stephen R. Marschner, “Microfacet models refraction through rough surfaces”, pp. 195–206, 2007.
- [13] J. Hasselgren, N. Hofmann, and J. Munkberg, “Shape, Light, and Material Decomposition from Images using Monte Carlo Rendering and Denoising”, *arXiv:2206.03380*, 2022.
- [14] A. T. Áfra, “Intel\textsuperscript{®} Open Image Denoise”. 2023.
- [15] [Online]. Available: <https://github.com/NVlabs/instant-ngp/blob/master/scripts/colmap2nerf.py>