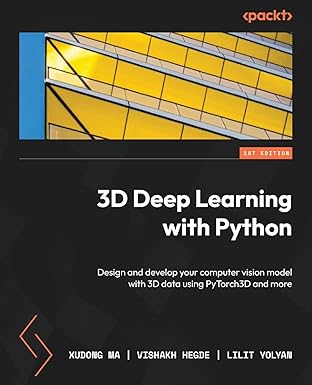
**3D Deep Learning with Python** - 2022

*Pack Publishing*



# PART 1: 3D Data Processing Basics

## Chapter 1: Introducing 3D Data Processing

### Technical requirements

The code snippets for this chapter can be found at

<https://github.com/PacktPublishing/3D-Deep-Learning-with-Python>

### Setting up a development environment

Installing Anaconda for Linux:

curl -O <https://repo.anaconda.com/archive/Anaconda3-2024.10-1-Linux-x86_64.sh>

bash ~/Anaconda3-2024.10-1-Linux-x86\_64.sh

After settingup Anaconda, run the following command to create a virtual environment of Python 3.7:

$ conda create -n python3d python=3.7

(*python3d* is just a random name we picked)

Activate the newly created virtual environments with the following command:

$ source activate python3d

3. Install PyTorch. Detailed instructions on installing PyTorch can be found on its web page at

<https://www.pytorch.org/get-started/locally/>

For example, I will install PyTorch 1.9.1 on my Ubuntu desktop with CUDA 11.8, as follows:

$ conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia

PyTorch3D may need some dependencies, and detailed instructions on how to install these dependencies can be found on the PyTorch3D GitHub home page at <https://github.com/facebookresearch/pytorch3d>

installing PyTorch3D can be easily done by running the following command:

$ conda install pytorch3d -c pytorch3d

### 3D data representation

#### Understanding point cloud representation

A 3D point cloud is a very straightforward representation of 3D objects, where each point cloud is just a collection of 3D points, and each 3D point is represented by one three-dimensional tuple (*x*, *y*, or *z*). The raw measurements of many depth cameras are usually 3D point clouds.

Unlike regular images, where we can define neighboring pixels for each individual pixel, there are no clear and regular definitions for neighboring points for each point in a point cloud – that is, convolutions usually cannot be applied to point clouds. Thus, special types of deep learning models need to be used for processing point clouds, such as PointNet: <https://arxiv.org/abs/1612.00593>.

#### Understanding mesh representation

#### Understanding voxel representation

A **voxel** is the counterpart of a pixel in 3D computer vision. A pixel is defined by dividing a rectangle in 2D into smaller rectangles and each small rectangle is one pixel. Similarly, a voxel is defined by dividing a 3D cube into smaller-sized cubes and each cube is called one voxel.

Voxel representations usually use **Truncated Signed Distance Functions** (**TSDFs**) to represent 3D surfaces. A **Signed Distance Function** (**SDF**) can be defined at each voxel as the (signed) distance between the center of the voxel to the closest point on the surface. A positive sign in an SDF indicates that the voxel center is outside an object.

### 3D data file format - Ply files

the two most frequently used data file formats to represent point clouds and meshes, the PLY file format and the OBJ file format.

An example, a cube.ply file, is shown in the following code snippet:

ply

format ascii 1.0

comment created for the book 3D Deep Learning with Python

element vertex 8

property float32 x

property float32 y

property float32 z

element face 12

property list uint8 int32 vertex\_indices

end\_header

-1 -1 -1

1 -1 -1

3 3 4 0

3 7 2 6

3 5 0 4

The **element vertex 8** line means that the first type of data in the PLY file is vertex and we have eight vertices. **property float32 x** means that each vertex has a property named x of the **float32 type**. Similarly, each vertex also has y and z properties. Here, each vertex is one 3D point. The **element face 12** line means that the second type of data in this PLY file is of the **face** type and we have 12 faces. **property list unit8 int32 vertex\_indices** shows that each face will be a list of vertex indices. The header part of the **ply** file always ends with an **end\_header** line.

The following is a code snippet, ply\_example1.py, for visualizing the mesh in the cube.ply file and loading the vertices and meshes as PyTorch tensors:

import open3d

from pytorch3d.io import load\_ply

mesh\_file = "cube.ply"

print('visualizing the mesh using open3D')

mesh = open3d.io.read\_triangle\_mesh(mesh\_file)

open3d.visualization.draw\_geometries([mesh],

mesh\_show\_wireframe = True,

mesh\_show\_back\_face = True)

print("Loading the same file with PyTorch3D")

vertices, faces = load\_ply(mesh\_file)

print('Type of vertices = ', type(vertices))

print("type of faces = ", type(faces))

print('vertices = ', vertices)

print('faces = ', faces)

### 3D data file format - OBJ files

Like the PLY file format, the OBJ format also has both an ASCII version and a binary version.

The first line, mtlib ./cube.mtl, declares the companion **Material Template Library** (**MTL**) file. The **MTL** file describes surface shading properties,

For the **o cube** line, the starting letter, o, indicates that the line defines an **object**, where the name of the object is cube.

Any line starting with # is a comment.

Each line starts with **v**, which indicates that each line defines a **vertex**. For example, v -0.5 -0.5 0.5 defines a vertex with an x coordinate of 0.5, a y coordinate of 0.5, and a z coordination of 0.5.

For each line starting with **f**, f indicates that each line contains a definition for one **face**. For example, the f 1 2 3 line defines a face, with its three vertices being the vertices with indices 1, 2, and 3.

The usemtl Door line declares that the surfaces declared after this line should be shaded using a material property defined in the MTL file, named Door:

mtllib ./cube.mtl

o cube

# Vertex list

v -0.5 -0.5 0.5

v -0.5 -0.5 -0.5

v -0.5 0.5 -0.5

v -0.5 0.5 0.5

v 0.5 -0.5 0.5

# Point/Line/Face list

usemtl Door

The cube.mtl companion MTL file is shown as follows. The file defines a material property called Door:

newmtl Door

Ka 0.8 0.6 0.4

Kd 0.8 0.6 0.4

Ks 0.9 0.9 0.9

d 1.0

Ns 0.0

illum 2

properties:

* Ka: Specifies an ambient color
* Kd: Specifies a diffuse color
* Ks: Specifies a specular color
* Ns: Defines the focus of specular highlights
* Ni: Defines the optical density (a.k.a index of refraction)
* d: Specifies a factor for dissolve
* **illum**: Specifies an illumination model
* **map\_Kd**: Specifies a color texture file to be applied to the diffuse reflectivity of the material

The cube.obj file can be opened by both Open3D and PyTorch3D.

print("Loading the same file with PyTorch3D")

vertices, faces, aux = load\_obj(mesh\_file)

### Understanding 3D coordination systems

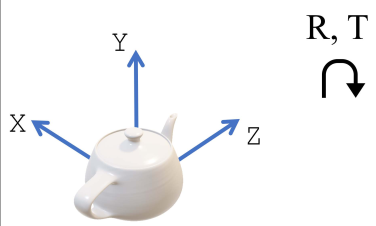


Figure 1.2 – A world coordinate system, where the origin and axis are defined independently of the camera positions

The first coordination system we frequently use is called the **world coordination system**. This coordinate system is a 3D coordination system chosen with respect to all the 3D objects, such that the locations of the 3D objects can be easy to determine. Usually, the axis of the world coordination system does not agree with the object orientation or camera orientation. Thus, there exist some non-zero rotations and displacements between the origin of the world coordination system and the object and camera orientations.

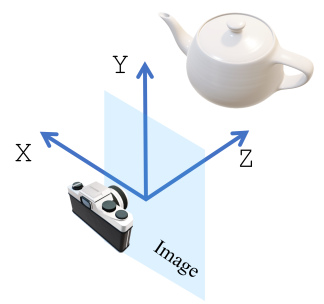


Figure 1.3 – The camera view coordinate system, where the origin is at the camera projection center and the three axes are defined according to the imaging plane

### Understanding camera models

In 3D deep learning, usually we need to use 2D images for 3D detection. Either 3D information is detected solely from 2D images, or 2D images are fused with depth for high accuracy. Nevertheless, camera models are essential to build correspondence between the 2D space and the 3D world.

In PyTorch3D, there are two major camera models, the orthographic camera defined by the OrthographicCameras class and the perspective camera model defined by the PerspectiveCameras class. The following figure shows the differences between the two camera models.

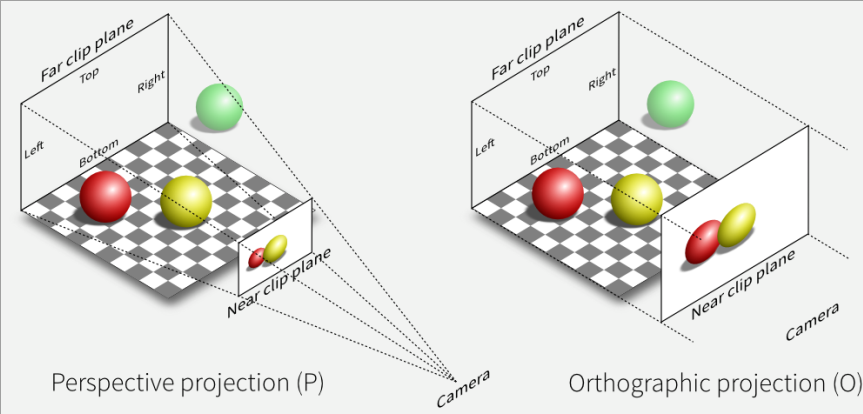


Figure 1.5 – Two major camera models implemented in PyTorch3D, perspective and orthographic

### Coding for camera models and coordination systems

3. We can load and visualize the mesh by using Open3D’s **draw\_geometrics** function:

#Load meshes and visualize it with Open3D

mesh\_file = "cube.obj"

print('visualizing the mesh using open3D')

mesh = open3d.io.read\_triangle\_mesh(mesh\_file)

open3d.visualization.draw\_geometries([mesh], mesh\_show\_wireframe = True,

mesh\_show\_back\_face = True)

4. We define a camera variable as a PyTorch3D PerspectiveCamera object. The camera

here is actually mini-batched. For example, the rotation matrix, R, is a PyTorch tensor with

a shape of [8, 3, 3], which actually defines eight cameras, each with one of the eight rotation

matrices.

#Define a mini-batch of 8 cameras

image\_size = torch.ones(8, 2)

image\_size[:,0] = image\_size[:,0] \* 1024

image\_size[:,1] = image\_size[:,1] \* 512

image\_size = image\_size.cuda()

focal\_length = torch.ones(8, 2)

focal\_length[:,0] = focal\_length[:,0] \* 1200

focal\_length[:,1] = focal\_length[:,1] \* 300

focal\_length = focal\_length.cuda()

principal\_point = torch.ones(8, 2)

principal\_point[:,0] = principal\_point[:,0] \* 512

principal\_point[:,1] = principal\_point[:,1] \* 256

principal\_point = principal\_point.cuda()

R = Rotation.from\_euler('zyx',

[[n\*5, n, n] for n in range(-4, 4, 1)],

degrees=True).as\_matrix()

R = torch.from\_numpy(R).cuda()

T = [ [n, 0, 0] for n in range(-4, 4, 1)]

T = torch.FloatTensor(T).cuda()

camera = PerspectiveCameras(focal\_length = focal\_length,

principal\_point = principal\_

point,

in\_ndc = False,

image\_size = image\_size,

R = R,

T = T,

device = 'cuda')

5. Once we have defined the camera variable, we can call the get\_world\_to\_view\_transform

class member method to obtain a Transform3d object, world\_to\_view\_transform. We

can then use the transform\_points member method to convert from world coordination

to camera view coordination.

world\_to\_view\_transform = camera.get\_world\_to\_view\_transform()

world\_to\_screen\_transform = camera.get\_full\_projection\_transform()

#Load meshes using PyTorch3D

vertices, faces, aux = load\_obj(mesh\_file)

vertices = vertices.cuda()

world\_to\_view\_vertices = world\_to\_view\_transform.transform\_points(vertices)

world\_to\_screen\_vertices = world\_to\_screen\_transform.transform\_points(vertices)

print('world\_to\_view\_vertices = ', world\_to\_view\_vertices)

print('world\_to\_screen\_vertices = ', world\_to\_screen\_vertices)

### Summary

## Chapter 2: Introducing 3D Computer Vision and Geometry

### Technical requirements

### Exploring the basic concepts of rendering, rasterization, and shading

the process of rendering can usually be divided into two stages – rasterization and shading. The ray tracing process is a typical rasterization process – that is, the process of finding relevant geometric objects for each image pixel. Shading is the process of taking the outputs of the rasterization and computing the pixel value for each image pixel.

The pytorch3d.renderer.mesh.rasterize\_meshes.rasterize\_meshes function in PyTorch3D usually computes the following four things for each image pixel:

* pix\_to\_face is a list of face indices that the ray may intersect.
* zbuf is a list of depth values of these faces.
* bary\_coords is a list of barycentric coordinates of the intersection point of each face and the ray.
* pix\_dists is a list of signed distances between pixels (*x* and *y*) and the nearest point on all the faces where the ray intersects. The values of this list can take negative values since it contains signed distances.

#### Understanding barycentric coordinates

For each point coplanar with a mesh face, the coordinates of the point can always be written as a linear combination of the coordinates of the three vertices of the mesh face. For example, as shown in the following diagram, the point p can be written as 𝑢A + 𝑣B + 𝑤C, where *A*, *B*, and *C* are the coordinates of the three vertices of the mesh face. Thus, we can represent each such point with the coefficients u, v, and w. This representation is called the **barycentric coordinates** of the point.

#### Light source models

#### Understanding the Lambertian shading model

The first physical model that we will discuss is Lambert’s cosine law. Lambertian surfaces are types of objects that are not shiny at all, such as paper, unfinished wood, and unpolished stones:

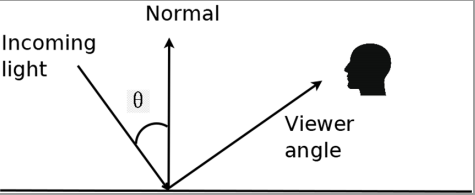
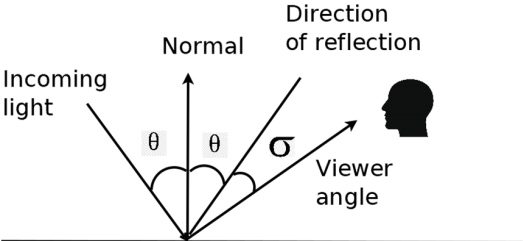


Figure 2.3: Light diffusion on Lambertian surfaces

#### Understanding the Phong lighting model

For shiny surfaces, such as polished tile floors and glossy paint, the reflected light also contains a highlight component. The Phong lighting model is a frequently used model for these glossy components:



### Coding exercises for 3D rendering

### Using PyTorch3D heterogeneous batches and PyTorch optimizers

- A coding exercise for a heterogeneous minibatch

### Understanding transformations and rotations

- A coding exercise for transformation and rotation

### Summary

# PART 2: 3D Deep Learning Using PyTorch3D

## Chapter 3: Fitting Deformable Mesh Models to Raw Point Clouds

### Technical requirements

### Fitting meshes to point clouds - the problem

Real-world depth cameras, such as LiDAR, time-of-flight cameras, and stereo vision cameras, usually output either **depth images** or **point clouds**.

**Loss functions** are central concepts in almost all optimizations. Essentially, to fit a point cloud, we need to design a loss function, such that when the loss function is minimized, the mesh as the optimization variable fits to the point cloud.

Actually, selecting the right loss function is usually a critical design decision in many real-world projects. Different choices of loss function usually result in significantly different system performance. The requirements for a loss function usually include at least the following properties:

* The loss function needs to have desirable numerical properties, such as smooth, convex, without the issue of vanishing gradients, and so on
* The loss function (and its gradients) can be easily computed; for example, they can be efficiently computed on GPUs
* The loss function is a good measurement of model fitting; that is, minimizing the loss function results in a satisfactory mesh model fitting for the input point clouds

### Formulating a deformable mesh fitting problem into an optimization problem

The cost function should be chosen such that it is a good measurement of how similar the point cloud is to the mesh.

### Loss functions for regularization

Machine learning literature has provided solutions for excluding such undesirable non-smooth solutions for several decades. The solution is called **regularization**. Essentially, the loss we want to optimize is chosen to be a sum of multiple loss functions. Certainly, the first term of the sum will be the primary **Chamfer** distance. The other terms are for penalizing surface non-smoothness and norm non-smoothness.

#### Mesh Laplacian smoothing loss

#### Mesh normal consistency loss

#### Mesh edge loss

### Implementing the mesh fitting with PyTorch3D

The input point cloud is contained in pedestrian.ply. The mesh can be visualized using the vis\_input.py code snippet. The main code snippet for fitting a mesh model to the point cloud is contained in deform1.py:

1. We will start by importing the needed packages:

import os

import sys

import torch

from pytorch3d.io import load\_ply, save\_ply

from pytorch3d.io import load\_obj, save\_obj

from pytorch3d.structures import Meshes

from pytorch3d.utils import ico\_sphere

from pytorch3d.ops import sample\_points\_from\_meshes

from pytorch3d.loss import (

chamfer\_distance,

mesh\_edge\_loss,

mesh\_laplacian\_smoothing,

mesh\_normal\_consistency,

)

import numpy as np

1. We then declare a PyTorch device. If you have GPUs, then the device would be created to use GPUs. Otherwise, the device has to use CPUs:

if torch.cuda.is\_available():

device = torch.device("cuda:0")

else:

device = torch.device("cpu")

print("WARNING: CPU only, this will be slow!")

1. We will load the point cloud from pedestrian.ply. Now, load\_ply is a PyTorch3D function that loads the .ply file and outputs verts and faces. In this case, verts is a PyTorch tensor. faces is an empty PyTorch tensor because pedestrian.ply actually does not contain any faces. The to member function moves the tensors to the device; if the device uses GPUs, then verts and faces are transmitted to the GPU memories:

verts, faces = load\_ply("pedestrian.ply")

verts = verts.to(device)

faces = faces.to(device)

1. We then run some normalization and change the tensor shapes for later processing:

center = verts.mean(0)

verts = verts - center

scale = max(verts.abs().max(0)[0])

verts = verts / scale

verts = verts[None, :, :]

1. In the next step, we create a mesh variable called src\_mesh by using the ico\_sphere PyTorch3D function. The ico\_sphere function essentially creates a mesh representing roughly a sphere. This src\_mesh will be our optimization variable; it will start as a sphere and then be optimized to fit the point cloud:

src\_mesh = ico\_sphere(4, device)

1. In the next step, we want to define a deform\_verts variable. deform\_verts is a tensor of vertex displacements, where for each vertex in src\_mesh, there is a vertex displacement of the three-dimensional vector. We are going to optimize deform\_verts for finding the optimal deformable mesh:

src\_vert = src\_mesh.verts\_list()

deform\_verts = torch.full(src\_vert[0].shape, 0.0,

device=device, requires\_grad=True)

1. We define an SGD optimizer with deform\_verts as the optimization variable:

optimizer = torch.optim.SGD([deform\_verts], lr=1.0, momentum=0.9)

1. We define a batch of weights for different loss functions. As we have mentioned, we need multiple loss functions, including the primary one and the regularization loss functions. The final loss will be a weighted sum of the different loss functions. Here is where we define the weights:

w\_chamfer = 1.0

w\_edge = 1.0

w\_normal = 0.01

w\_laplacian = 0.1

1. We are then ready for going into the major optimization iterations. We are going to iterate 2,000 times for computing the loss function, computing the gradients, and going along the gradient descent directions. Each iteration starts with optimizer.zero\_grad()

for i in range(0, 2000):

print("i = ", i)

optimizer.zero\_grad()

new\_src\_mesh = src\_mesh.offset\_verts(deform\_verts)

sample\_trg = verts

sample\_src = sample\_points\_from\_meshes(new\_src\_mesh, verts.shape[1])

loss\_chamfer, \_ = chamfer\_distance(sample\_trg, sample\_src)

loss\_edge = mesh\_edge\_loss(new\_src\_mesh)

loss\_normal = mesh\_normal\_consistency(new\_src\_mesh)

loss\_laplacian = mesh\_laplacian\_smoothing(new\_src\_mesh, method="uniform")

loss = (

loss\_chamfer \* w\_chamfer

+ loss\_edge \* w\_edge

+ loss\_normal \* w\_normal

+ loss\_laplacian \* w\_laplacian

)

loss.backward()

optimizer.step()

1. We then extract the obtained vertices and faces from the new\_src\_mesh variable and then resume its original center location and scale:

final\_verts, final\_faces = new\_src\_mesh.get\_mesh\_verts\_faces(0)

final\_verts = final\_verts \* scale + center

1. Finally, the obtained mesh model is saved in the deform1.ply file:

final\_obj = os.path.join("./", "deform1.ply")

save\_ply(final\_obj, final\_verts, final\_faces, ascii=True)

#### The experiment of not using any regularization loss functions

#### The experiment of using only the mesh edge loss

### Summary

## Chapter 4: Learning Object Pose Detection and Tracking by Differentiable Rendering

**Rendering** is the process of projecting 3D physical models (a mesh model for the object, or a camera model) into 2D images. It is an imitation of the physical process of image formation. Many 3D computer vision tasks can be considered as an inverse of the rendering process – that is, in many computer vision problems, we want to start from 2D images to estimate the 3D physical models (meshes, point cloud segmentation, object poses, or camera positions)

### Technical requirements

### Why we want to have differentiable rendering

### How to make rendering differentiable

#### What problems can be solved by using differentiable rendering

### The object pose estimation problem

### How it is coded

- An example of object pose estimation for both silhouette fitting and texture fitting

### Summary

## Chapter 5: Understanding Differentiable Volumetric Rendering

We are going to use a voxel 3D data representation, unlike the mesh 3D data representation we used in the last chapter. **Voxel** 3D data representation has certain advantages compared to mesh models. For example, it is more flexible and highly structured.

### Technical requirements

### Overview of volumetric rendering

The main goal of volumetric rendering is to render a 2D projection of 3D data since that is what our eyes can perceive on a flat screen.

### Understanding ray sampling

### Using volume sampling

Volume sampling is the process of getting color and occupancy information along the points provided by the ray samples.

### Exploring the ray marcher

Now that we have the color and density values for all the points sampled with the ray sampler, we need to figure out how to use it to finally render the pixel value on the projected image. In this section, we are going to discuss the process of converting the densities and colors on points of rays to RGB values on images. This process models the physical process of image formation.

### Differentiable volumetric rendering

While standard volumetric rendering is used to render 2D projections of 3D data, differentiable volume rendering is used to do the opposite: construct 3D data from 2D images.

#### Reconstructing 3D models from multi-view images

### Summary

## Chapter 6: Exploring Neural Radiance Fields (NeRF)

### Technical requirements

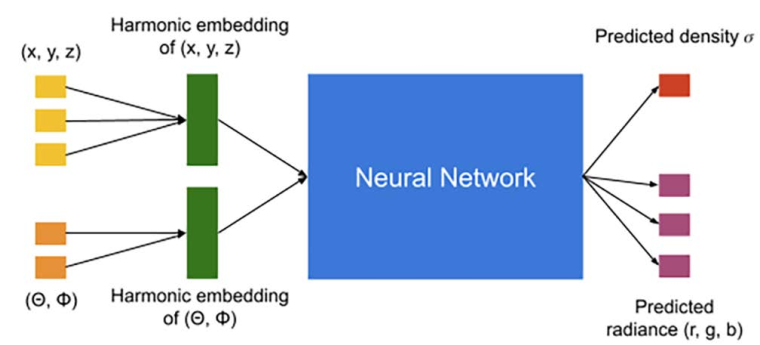
### Understanding NeRF

#### What is a radiance field?

Radiance is the standard metric for measuring the amount of light that passes through or is emitted from an area inside a particular solid angle. For our purposes, we can treat the radiance to be the intensity of a point in space when viewed in a particular direction.

#### Representing radiance fields with neural networks

NeRF uses a neural network to represent a volumetric scene function.



### Training a NeRF model

### Understanding the NeRF model architecture

### Understanding volume rendering with radiance fields

- Projecting rays into the scene

- Accumulating the color of a ray

### Summary

# PART 3: State-of-the-art 3D Deep Learning Using PyTorch3D

## Chapter 7: Exploring Controllable Neural Feature Fields

### Technical requirements

### Understanding GAN-based image synthesis

### Introducing compositional 3D-aware image synthesis

Our goal is controllable image synthesis. We need control over the number of objects in the image, their position, shape, size, and pose. The GIRAFFE model is one of the first to achieve all these desirable properties while also generating high-resolution photorealistic images. In order to have control over these attributes, the model must have some awareness of the 3D nature of the scene.

### Generating feature fields

The first step of the scene generation process is generating a feature field. This is analogous to generating an RGB image in the NeRF model.

### Mapping feature fields to images

### Exploring controllable scene generation

#### Exploring controllable car generation

#### Exploring controllable face generation

### Training the GIRAFFE model

#### Frechet Inception Distance

In order to evaluate the quality of generated images, we use the **Frechet Inception Distance** (**FID**). This is a measure of the distance between features extracted from real and generated images.

#### Training the model

### Summary

## Chapter 8: Modeling the Human Body in 3D

### Technical requirements

### Formulating the 3D modeling problem

- Defining a good representation

### Understanding the Linear Blend Skinning technique

### Understanding the SMPL model

- Defining the SMPL model

### Using the SMPL model

### Estimating 3D human pose and shape using SMPLify

The SMPLify approach consists of the following two stages:

1. Automatically detect 2D joints using established pose detection models such as OpenPose or DeepCut. Any 2D joint detectors can be used in their place as long as they are predicting the same joints.

2. Use the SMPL model to generate the 3D shape. Directly optimize the parameters of the SMPL model so that the model joints of the SMPL model project to the 2D joints predicted in the previous stage.

#### Defining the optimization objective function

### Exploring SMPLify

#### Running the code

#### Exploring the code

### Summary

## Chapter 9: Performing End-to-End View Synthesis with SynSin

This chapter is dedicated to the latest state-of-the-art view synthesis model called SynSin. View Synthesis is one of the main directions in 3D deep learning, which can be used in multiple different domains such as AR, VR, gaming, and more. The goal is to create a model for the given image as an input to reconstruct a new image from another view.

### Technical requirements

### Overview of view synthesis

### SynSin network architecture

#### Spatial feature and depth networks

#### Neural point cloud renderer

#### Refinement module and discriminator

### Hands-on model training and testing

### Summary

## Chapter 10: Mesh R-CNN

This chapter is dedicated to a state-of-the-art model called Mesh R-CNN, which aims to combine two different but important tasks into one end-to-end model. It is a combination of the well-known image segmentation model Mask R-CNN and a new 3D structure prediction model. These two tasks were researched a lot separately.

### Technical requirements

### Overview of meshes and voxels

### Mesh R-CNN architecture

#### Graph convolutions

#### Mesh predictor

### Demo of Mesh R-CNN with PyTorch

- Demo

### Summary

# Other Books You May Enjoy