Содержание

Об авторах

Ксудонг Ма – штатный инженер машинного обучения в Grabango Inc. Беркли, штат Калифорния. Работал старшим инженером машинного обучения в Facebook (Meta) Oculus и тесно сотрудничал с коллективом 3D PyTorch в проектах отслеживания лица в 3D. Имеет многолетний опыт работы в области компьютерного зрения, машинного и глубокого обучения и имеет докторскую степень в области электромашиностроения и конструирования вычислительных машин.

Вишах Хегде – исследователь в области машинного обучения и компьютерного зрения. Имеет более 7 лет опыта работы в указанных областях, во время которых стал автором нескольких хорошо процитированных исследовательских работ и опубликованных патентов. Имеет степень магистра Стэнфордского университета по специализации «Прикладная математика и машинное обучение», а также степени бакалавра и магистра по физике университета IIT в Мадрасе. Ранее работал в Schlumberger и Matroid. Является старшим прикладным исследователем в Ambient.ai, где помогал разрабатывать систему обнаружения оружия, которая развернута в нескольких глобальных компаниях из списка Fortune 500. Сейчас он использует свой опыт и страсть к решению деловых задач с целью создания технологического стартапа в Кремниевой долине. Подробнее о нем можно узнать на его сайте.

Хотел бы поблагодарить исследователей в области компьютерного зрения, прорывное исследование которых мне пришлось описывать. Хотел бы поблагодарить рецензентов за их отзыв и замечательный коллектив издательства Packt Publishing за то, что он дал мне возможность проявить творческий подход. Наконец, хочу поблагодарить свою жену и семью за их поддержку и вдохновение в ситуации, когда мне это было нужно больше всего.

Лилит Йольян – исследователь машинного обучения, работающая над докторской диссертацией в университете в YSU. Ее исследования посвящены разработке технологических решений в области компьютерного зрения для умных городов с использованием данных дистанционной съемки. Имеет 5-летний опыт работы в области компьютерного зрения и работала над технически сложным решением по обеспечению безопасности водителя, планируемым к развертыванию многими известными компаниями-производителями автомобилей.

Предисловие

Благодаря этому практическому руководству по трехмерному глубокому обучению разработчики в области трехмерного компьютерного зрения смогут применить свои знания на практике. В данной книге представлен практический подход к реализации вычислительных решений в указанной области и связанных с ней методологий, которые помогут вам быстро начать работу и повысить продуктивность.

Оснащенные пошаговыми объяснениями важных понятий, практическими примерами и вопросами для самопроверки, вы начнете с обследования передовых методов трехмерного глубокого обучения.

Вы познакомитесь с базовой обработкой 3D-данных полигональной сетки и облака точек с помощью библиотеки PyTorch3D, такой как загрузка и сохранение файлов PLY и OBJ, проецирование 3D-точек на координаты камеры с использованием моделей перспективной камеры и ортографической камеры, отрисовка облаков точек и полигональных сеток на изображениях и т. д. Вы также научитесь реализовывать некоторые современные алгоритмы трехмерного глубокого обучения, такие как дифференцируемая отрисовка, NeRF, SynSin и Mesh R-CNN, поскольку благодаря библиотеке PyTorch3D программирование этих моделей глубокого обучения значительно упрощается.

К концу этой книги вы сможете реализовывать свои собственные модели трехмерного глубокого обучения.

Для кого эта книга предназначена

Эта книга предназначена для всех тех, кто начинает свою карьеру в области машинного обучения, а также практиков среднего уровня, исследователей данных, инженеров машинного обучения и инженеров глубокого обучения, которые стремятся хорошо разбираться в методах компьютерного зрения, используя 3D-данные.

О чем эта книга рассказывает

Глава 1 «Введение в обработку 3D-данных» будет посвящена основам 3D-данных, например способам хранения 3D-данных и базовым понятиям полигональной сетки и облаков точек, мировой системы координат и системы

координат поля зрения камеры. В ней также дается объяснение часто используемой системы координат NDC, способов конверсии разных систем координат, перспективной и ортографической камер и моделей камеры, которые следует использовать.

Глава 2 «Введение в трехмерное компьютерное зрение и геометрию» покажет базовые понятия компьютерной графики, такие как отрисовка и затенение. Вы познакомитесь с несколькими фундаментальными понятиями, которые потребуются в последующих главах этой книги, включая 3D-трансформации геометрии, тензоры PyTorch и оптимизацию.

Глава 3 «Подгонка деформируемых сеточных моделей к необработанным облакам точек» представит практический проект применения деформируемой 3D-модели с целью подгонки шумных 3D-наблюдений, используя все знания, которые вы получили в предыдущих главах. Вы познакомитесь с часто используемыми функциями стоимости, причинами важности этих функций и ситуациями, когда эти функции стоимости обычно используются. Наконец, мы обследуем наглядный пример выбора конкретных функций стоимости под конкретную задачу и настройки цикла оптимизации, чтобы получить желаемые результаты.

Глава 4 «Обнаружение и отслеживание позы объекта с помощью дифференцируемой отрисовки» расскажет о базовых концепциях дифференцируемой отрисовки. Она поможет разобраться в основных понятиях и выяснить, в каких ситуациях следует эти методы применять для решения своих собственных задач.

Глава 5 «Понятие дифференцируемой объемометрической отрисовки» представит практический проект с использованием дифференцируемой отрисовки для оценивания позиций камеры по одному изображению и известной трехмерной сеточной модели. Вы научитесь применять библиотеку PyTorch3D на практике, чтобы настраивать камеры, отрисовщики и затенители. Вы также получите практический опыт использования разных функций стоимости, чтобы получать результаты оптимизации.

Глава 6 «Обследование нейронных полей яркости излучения (NeRF)» предоставит практический проект с использованием дифференцируемого отрисовщика для оценивания трехмерных сеточных моделей по нескольким изображениям и текстурным моделям.

Глава 7 «Обследование контролируемых нейронных полей признаков» охватывает очень важный алгоритм синтеза ракурсов под названием Nerf. Вы познакомитесь с тем, что это вообще такое, как его использовать и где он проявляет свою ценность.

Глава 8 «Моделирование человеческого тела в 3D» посвящена обследованию подгонки 3D-тела человека с использованием алгоритма SMPL.

Глава 9 «Сквозной синтез ракурсов с помощью модели SynSin» посвящена передовой модели глубокого обучения, применяемой для синтеза других ракурсов изображения.

Глава 10 «Модель Mesh R-CNN» познакомит еще с одним передовым методом предсказания трехмерных воксельных моделей по одному входному изображению под названием Mesh R-CNN.

Что нужно, чтобы извлечь максимум пользы из этой книги

Описанное в книге программное/аппаратное обеспечение	Требования к операционной системе
Python 3.6+	Windows, MacOS или Linux

Если вы используете цифровую версию этой книги, то советуем набирать исходный код самостоятельно либо обращаться к исходному коду в репозитории книги на GitHub (ссылка на репозиторий доступна в следующем разделе). Это поможет избежать любых потенциальных ошибок, связанных с копированием и вставкой исходного кода.

Для справки, пожалуйста, ознакомьтесь с перечисленными ниже статьями.

Глава 6: https://arxiv.org/abs/2003.08934, https://github.com/yenchenlin/nerf-pytorch.

Глава 7: https://m-niemeyer.github.io/project-pages/giraffe/index.html, https://arxiv.org/abs/2011.12100.

Глава 8: https://smpl.is.tue.mpg.de/, https://smplify.is.tue.mpg.de/, https://smpl-x.is.tue.mpg.de/.

Глава 9: https://arxiv.org/pdf/1912.08804.pdf.

Глава 10: https://arxiv.org/abs/1703.06870, https://arxiv.org/ abs/1906.02739.

Используемые обозначения

В этой книге используется ряд текстовых обозначений.

`Исходный код в тексте`: указывает слова исходного кода в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, имена путей, фиктивные URL-адреса, вводимые пользователем данные и дескрипторы Twitter. Например, «Далее нужно обновить файл `./options/options.py`».

Блок исходного кода задается, как показано ниже:

```
elif opt.dataset == 'kitti':
    opt.min_z = 1.0
    opt.max_z = 50.0
    opt.train_data_path = (
        './DATA/dataset_kitti/'
    )
    from data.kitti import KITTIDataLoader
    return KITTIDataLoader
```

Когда мы хотим привлечь ваше внимание к определенной части блока исходного кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
wget https://dl.fbaipublicfiles.com/synsin/checkpoints/realestate/synsin.pth
```

Любые данные на входе или на выходе из команды командой оболочки записываются, как показано ниже:

```
bash ./download_models.sh
```

Жирный шрифт: выделяет новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах пишутся в тексте следующим образом: «Модуль детализации (**g**) получает входные данные от нейронного отрисовщика облака точек и затем выводит окончательное реконструированное изображение».

Подсказки и важные замечания выглядят так.

Часть I

ОСНОВЫ ОБРАБОТКИ 3D-ДАННЫХ

Первая часть книги посвящена определению самых базовых понятий обработки данных и изображений, поскольку указанные понятия лягут в основу последующего изложения. Данная часть книги делает книгу самодостаточной, вследствие чего читателям не придется читать какие-либо другие книги, чтобы приступить к изучению библиотеки PyTorch3D.

Эта часть содержит следующие главы:

- глава 1 «Введение в обработку 3D-данных»;
- глава 2 «Введение в трехмерное компьютерное зрение и геометрию».

Глава 1

Введение в обработку 3D-данных

В этой главе мы обсудим несколько базовых понятий, весьма существенных для трехмерного глубокого обучения, которые будут часто использоваться в последующих главах. Вы начнете со знакомства с наиболее часто используемыми форматами 3D-данных, а также многими способами манипулирования ими и конвертации их в разные форматы. Мы начнем с настройки среды разработки и установкой всех необходимых программных пакетов, включая Anaconda, Python, PyTorch и PyTorch3D. Затем мы поговорим о наиболее часто используемых способах представления 3D-данных – например, облаках точек, полигональных сетках и вокселах. Затем мы перейдем к форматам файлов 3D-данных, таким как файлы PLY и OBJ. Затем обсудим системы 3D-координат. Наконец, мы обсудим модели камеры, которые в основном связаны со способом отображения 3D-данных в 2D-изображения[1].

После прочтения этой главы вы сможете легко отлаживать алгоритмы трехмерного глубокого обучения, проводя инспекцию файлов выходных данных. Благодаря четкому пониманию систем координат и моделей камеры вы будете готовы опираться на эти знания и узнать о более продвинутых темах трехмерного глубокого обучения.

В данной главе будут охвачены следующие ниже главные темы:

- настройка среды разработки и установка дистрибутива Anaconda, библиотек PyTorch и PyTorch3D,
- представление 3D-данных,
- форматы 3D-данных – файлы PLY и OBJ,
- системы 3D-координат и конверсия между ними,
- модели камеры – перспективная и ортографическая камеры.

[1] Син. соотнесение 3D-данных с 2D-изображениями. – *Прим. перев.*

Технические требования

Для выполнения примеров исходного кода этой книги в идеале понадобится компьютер с графическим процессором. Тем не менее для выполнения фрагментов исходного кода вполне будет достаточно только центрального процессора(ов).

Рекомендуемая компьютерная конфигурация включает следующее:

- GPU, такой как серия GTX или серия RTX с не менее 8 Гб памяти,
- Python 3,
- библиотеки PyTorch и PyTorch3D.

Фрагменты исходного кода к этой главе находятся по адресу https://github.com/packtpublishing/3d-deep-learning-with-python.

Настройка среды разработки

Сначала давайте создадим среду разработки для всех прилагаемых к этой книге примеров программирования. Для всех примеров исходного кода Python в этой книге рекомендуется использовать машину Linux.

1. Сначала мы настроим широко используемый дистрибутив Python под названием Anaconda, который идет в комплекте с мощной реализацией CPYTHON. Одним из преимуществ использования дистрибутива Anaconda является его система управления пакетами, позволяющая пользователям легко создавать виртуальные среды. Индивидуальная редакция дистрибутива является бесплатной для одиночных практиков, студентов и исследователей. В целях установки дистрибутива мы рекомендуем посетить его веб-сайт anaconda.com, на котором можно получить подробные инструкции. Самым простым способом установки дистрибутива Anaconda, как правило, является выполнение скрипта, который нужно скачать с веб-сайта дистрибутива. После настройки дистрибутива выполните следующую ниже команду, чтобы создать виртуальную среду Python 3.7:

   ```
   $ conda create -n python3d python=3.7
   ```

   Эта команда создаст виртуальную среду Python версии 3.7. Для того чтобы использовать эту виртуальную среду, ее нужно сначала активировать.
2. Активируйте только что созданную виртуальную среду следующей ниже командой:

   ```
   $ source activate python3d
   ```

3. Установите библиотеку PyTorch. Подробные инструкции по установке PyTorch находятся на ее веб-странице по адресу www.pytorch.org/get-

started/locally/. Например, на своем рабочем столе Ubuntu с CUDA 11.1 я установлю PyTorch 1.9.1 следующим образом:

```
$ conda install pytorch torchvision torchaudio
cudatoolkit-11.1 -c pytorch -c nvidia
```

4. Установите библиотеку PyTorch3D. Это библиотека Python с открытым исходным кодом для трехмерного компьютерного зрения, недавно выпущенная исследовательской группой Facebook AI Research. Библиотека PyTorch3D предоставляет много функций-утилит, позволяющих с легкостью манипулировать 3D-данными. Будучи спроектированной специально для глубокого обучения, она позволяет обрабатывать почти все 3D-данные мини-пакетами, такими как камеры, облака точек и полигональные сетки. Еще одной ключевой особенностью библиотеки PyTorch3D является реализация очень важной техники трехмерного глубокого обучения, именуемой *дифференцируемой отрисовкой*[1]. Тем не менее самым большим преимуществом данной библиотеки трехмерного глубокого обучения является ее тесная связь с PyTorch.
   Библиотеке PyTorch3D могут понадобиться некоторые зависимости, и подробные инструкции по установке этих зависимостей находятся на домашней странице PyTorch3D на Github по адресу github.com/facebookresearch/pytorch3d. После установки всех зависимостей, если следовать инструкциям веб-сайта, установка PyTorch3D легко выполняется следующей ниже командой:

```
$ conda install pytorch3d -c pytorch3d
```

Теперь, когда мы создали среду разработки, давайте продолжим и займемся изучением представления данных.

Представление 3D-данных

В этом разделе вы познакомитесь с наиболее часто используемыми представлениями 3D-данных. Выбор представления данных является особенно важным конструктивным решением для многих систем трехмерного глубокого обучения. Например, облака точек не имеют решетчатых структур, поэтому свертки обычно невозможно использовать для них напрямую. Представления в виде вокселов имеют решетчатые структуры, однако они, как правило, потребляют большой объем компьютерной памяти. Мы обсудим плюсы и минусы этих 3D-представлений подробнее в этом разделе. Пред-

[1] Дифференцируемая отрисовка (differentiable rendering) – это относительно новая и захватывающая область исследований в компьютерном зрении, преодолевающая разрыв между 2D и 3D, позволяющая связывать пикселы 2D-изображения с 3D-свойствами сцены. – *Прим. перев.*

ставлениями 3D-данных, получившими наиболее широкое применение на практике, обычно являются облака точек, полигональные сетки и вокселы.

Представление в виде облака точек

Облако 3D-точек – это очень простое представление 3D-объектов, в котором каждое облако точек – это просто множество 3D-точек, и каждая 3D-точка представлена одним трехмерным кортежем (*x*, *y* и *z*). Сырые мерные данные многих камер глубины обычно представляют собой трехмерные облака точек.

С точки зрения глубокого обучения облака 3D-точек являются одним из неупорядоченных и нерегулярных типов данных. В отличие от регулярных изображений, в которых по каждому отдельному пикселу можно определить соседствующие ему пикселы, в облаке точек нет четких и регулярных определений соседних точек по каждой точке – т. е. применить свертки к облакам точек обычно невозможно. И поэтому для обработки облаков точек необходимо использовать специальные типы моделей глубокого обучения, такие как PointNet: https://arxiv.org/abs/1612.00593.

Еще одной проблемой облаков точек в качестве тренировочных данных для трехмерного глубокого обучения является проблема разнородности данных – т. е. по каждому тренировочному набору данных разные облака точек могут содержать разное число 3D-точек. Один из подходов к решению проблемы разнородности данных заключается в вынужденном назначении всем облакам точек одинакового числа точек. Однако это не всегда возможно – например, число возвращаемых камерами глубины точек может отличаться от кадра к кадру.

При тренировке моделей глубокого обучения разнородные данные могут создавать некоторые трудности для мини-пакетного градиентного спуска. В большинстве систем глубокого обучения подразумевается, что каждый мини-пакет содержит тренировочные примеры одинакового размера и мерности. Предпочитаются именно такие однородные данные, поскольку их можно обрабатывать на современном оборудовании для параллельной обработки наиболее эффективным образом, таком как графические процессоры. Эффективная обработка разнородных мини-пакетов требует дополнительной работы. К счастью, PyTorch3D предоставляет целый ряд способов эффективной обработки разнородных мини-пакетов, которые очень важны для трехмерного глубокого обучения.

Представление в виде полигональной сетки

Полигональные сетки, или меши, – это еще одно широко используемое представление 3D-данных. Как и точки в облаках точек, каждая полигональная сетка содержит множество 3D-точек, именуемых вершинами. Кроме того, каждая полигональная сетка содержит множество многоугольников, именуемых гранями, которые определены на вершинах.

В большинстве основанных на данных приложений полигональные сетки являются результатом постобработки сырых мерных данных камер глубины. Нередко они создаются вручную в процессе конструирования 3D-ресурсов. По сравнению с облаками точек полигональные сетки содержат дополнительную геометрическую информацию, кодируют топологию и имеют информацию о нормалях к поверхности. Эта дополнительная информация становится особенно полезной в тренировке обучающихся моделей. Например, графовые сверточные нейронные сети обычно трактуют полигональные сетки как графы и определяют сверточные операции, используя информацию о соседстве вершин.

Подобно облакам точек, полигональные сетки также имеют схожие проблемы разнородности данных. И снова PyTorch3D предоставляет эффективные способы оперирования разнородными мини-пакетами данных полигональной сетки, что делает трехмерное глубокое обучение весьма эффективным.

Представление в виде воксела

Еще одним важным представлением 3D-данных является представление в виде воксела. Воксел – это аналог пиксела в трехмерном компьютерном зрении. Пиксел определяется путем деления прямоугольника в 2D на меньшие прямоугольники, и каждый малый прямоугольник – это один пиксел. По аналогии с этим воксел определяется путем деления трехмерного куба на кубы меньшего размера, и каждый такой куб называется одним вокселом. Соответствующие процессы показаны на следующем ниже рисунке:

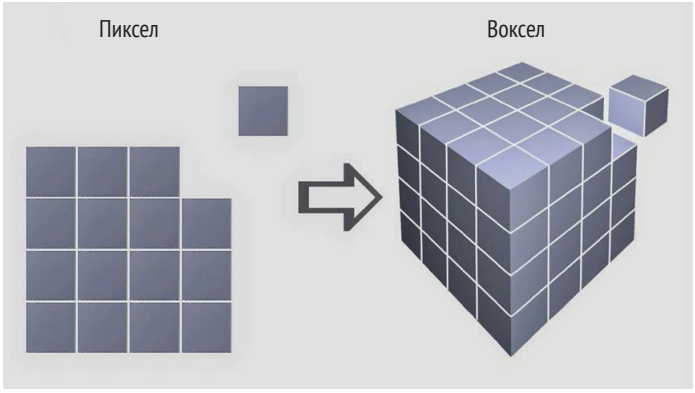

Рис. 1.1 ❖ Представление в виде воксела является трехмерным аналогом двумерного представления в виде пиксела, где кубическое пространство делится на малообъемные элементы

В представлениях в виде воксела для представления 3D-поверхностей обычно используются **функции усеченных расстояний со знаком** (**TSDF**)[1].

[1] От англ. *Truncated Signed Distance Function. – Прим. перев.*

Функция расстояния со знаком (**SDF**)[1] может быть определена на каждом вокселе в качестве расстояния (со знаком) между центром воксела до ближайшей точки на поверхности. Положительный знак в SDF указывает на то, что центр воксела находится вне объекта. Единственное различие между TSDF и SDF заключается в том, что значения TSDF усекаются, в результате чего значения TSDF всегда варьируются в интервале от –1 до +1.

В отличие от представлений в виде облаков точек и полигональных сеток представление в виде вокселов упорядочено и является регулярным. Это свойство похоже на пикселы в изображениях и позволяет использовать сверточные фильтры в моделях глубокого обучения. Одним из потенциальных недостатков представления в виде вокселов является то, что для них обычно требуется больше компьютерной памяти, но указанный недостаток можно уменьшить за счет таких методов, как хеширование. Тем не менее представление в виде вокселов является важным представлением 3D-данных.

Существуют представления 3D-данных, отличные от упомянутых выше. Например, в многоракурсных представлениях используется несколько изображений, взятых с разных точек зрения, чтобы представлять трехмерную сцену. В представлениях RGB-D используется дополнительный канал глубины, чтобы представлять 3D-сцену. Однако в этой книге мы не будем погружаться в эти 3D-представления слишком глубоко. Теперь, когда вы познакомились с основами представлений 3D-данных, самое время заняться несколькими форматами файлов, часто используемыми для облаков точек и полигональных сеток.

Формат файла 3D-данных – файлы PLY

Формат файла PLY[2] был разработан в середине 1990-х годов группой исследователей из Стэнфордского университета. С тех пор он превратился в один из наиболее широко используемых форматов файлов 3D-данных. Формат файла PLY имеет как ASCII-версию, так и двоичную версию. Двоичная версия предпочтительнее в тех случаях, когда необходимо минимизировать размеры файлов и обеспечить эффективность обработки. ASCII-версию легко отлаживать. Здесь мы обсудим базовый формат PLY-файлов и технику использования как пакета Open3D, так и библиотеки PyTorch3D для загрузки и визуализации 3D-данных из PLY-файлов.

В этом разделе мы собираемся обсудить два наиболее часто используемых формата файлов данных, чтобы представлять облака точек и полигональные сетки, формат PLY-файла и формат OBJ-файла. Мы обсудим сами форматы и способы загрузки и сохранения этих форматов файлов с помощью библиотеки PyTorch3D. Библиотека PyTorch3D предоставляет отличные функции-утилиты, поэтому с помощью этих утилит загрузка из этих форматов и сохранение в них проста и эффективна.

---

[1] От англ. *Signed Distance Function. – Прим. перев.*

[2] От англ. *Polygon File Format. – Прим. перев.*

Пример PLY-файла `cube.ply` показан в следующем ниже фрагменте исходного кода (данный файл находится в папке `ply_io` главы книги в репозитории на GitHub):

```
ply
format ascii 1.0
comment создан для книги Глубокое обучение в 3-D на 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
1 1 -1
-1 1 -1
-1 -1 1
1 -1 1
1 1 1
-1 1 1
3 0 1 2
3 5 4 7
3 6 2 1
3 3 7 4
3 7 3 2
3 5 1 0
3 0 2 3
3 5 7 6
3 6 1 5
3 3 4 0
3 7 2 6
3 5 0 4
```

Как видно из приведенного выше примера, каждый PLY-файл содержит секцию заголовка и секцию данных. Первая строка каждого PLY-файла в кодировке ASCII всегда содержит ключевое слово `ply`, указывая на то, что это PLY-файл. Вторая строка `format ascii 1.0` показывает, что файл имеет кодировку ASCII с номером версии. Любые строки, начинающиеся с ключевого слова `comment`, будут рассматриваться как строка комментариев, и, следовательно, все, что следует за комментарием, будет игнорироваться при загрузке PLY-файла компьютером. Строка `element vertex 8` означает, что первым типом данных в PLY-файле является вершина, и всего имеется восемь вершин. Выражение `property float32 x` означает, что в каждой вершине есть свойство `x` с типом `float32`. Аналогичным образом каждая вершина также имеет свойства `y` и `z`. Здесь каждая вершина – это одна 3D-точка. Строка `element face 12` означает, что второй тип данных в указанном PLY-файле имеет тип `face`, и всего имеется 12 граней. Выражение `property list uint8 int32 vertex_indices` показывает, что каждая грань будет списком индексов вершин. Секция заголовка PLY-файла всегда заканчивается строкой `end_header`.

Первая часть секции данных PLY-файла состоит из восьми строк, каждая строка которой является записью одной вершины. Три числа в каждой строке представляют три свойства `x`, `y` и `z` вершины. Например, три числа `-1`, `-1`, `-1` указывают на то, что координата `x` вершины равна `-1`, координата `y` вершины равна `-1` и координата `z` вершины равна `-1`.

Вторая часть секции данных PLY-файла состоит из 12 строк, каждая строка которой является записью одной грани. Первое число в последовательности чисел указывает число имеющихся у грани вершин, а последующие числа являются индексами вершин. Индексы вершин определяются по порядку, в котором вершины объявлены в PLY-файле.

Для открытия приведенного выше файла можно использовать как пакет Open3D, так и библиотеку PyTorch3D. Пакет Python Open3D очень удобен для визуализации 3D-данных, а библиотека PyTorch3D удобна для применения этих данных в моделях глубокого обучения. Ниже приведен фрагмент исходного кода файла Python `ply_example1.py` в папке `ply_io` главы книги в репозитории на GitHub для визуализации полигональной сетки в PLY-файле `cube.ply` и загрузки вершин и полигональной сетки в виде тензоров PyTorch:

```
import open3d
from pytorch3d.io import load_ply

mesh_file = 'cube.ply'
print('Визуализация полигональной сетки с помощью Open3D')
mesh = open3d.io.read_triangle_mesh(mesh_file)
open3d.visualization.draw_geometries([mesh],
       mesh_show_wireframe = True,
       mesh_show_back_face = True)

print('Загрузка того же файла с помощью PyTorch3D')
vertices, faces = load_ply(mesh_file)
print('Тип vertices = ', type(vertices))
print('Тип faces = ', type(faces))
print('vertices = ', vertices)
print('faces = ', faces)
```

В приведенном выше фрагменте исходного кода Python PLY-файл полигональной сетки `cube.ply` сначала открывается с помощью функции `read_triangle_mesh`[1] пакета Open3D, и все 3D-данные читаются в переменную `mesh`. Затем полигональную сетку можно визуализировать, используя функцию `draw_geometries`[2] данного пакета. При выполнении этой функции пакет Open3D выведет на экран окно для интерактивной визуализации полигональной сетки, в котором ее можно интерактивно поворачивать, увеличивать и уменьшать ее масштаб, используя мышь. PLY-файл `cube.ply`, как можно догадаться, определяет полигональную сетку куба с восемью вершинами и шестью сторонами, каждая сторона которого покрыта двумя гранями.

---

[1] Прочитать треугольную сетку. – *Прим. перев.*

[2] Начертить геометрические объекты. – *Прим. перев.*

Для загрузки той же сетки также можно использовать библиотеку PyTorch3D. Однако на этот раз мы собираемся получить несколько тензоров PyTorch – например, один тензор для вершин и один тензор для граней. Эти тензоры можно напрямую водить в любую модель глубокого обучения PyTorch. В данном примере функция `load_ply` возвращает кортеж с вершинами и гранями, обе из которых обычно находятся в формате тензоров PyTorch. При выполнении исходного кода файла Python `ply_example1.py` возвращенные вершины должны быть тензором PyTorch с очертанием[1] `[8, 3]`, т. е. восемь вершин, и в каждой вершине по три координаты. Аналогичным образом возвращенные грани должны быть тензором PyTorch с очертанием `[12, 3]`, т. е. 12 граней, и у каждой грани по 3 индекса вершин.

В следующем ниже фрагменте исходного кода демонстрируется еще один пример PLY-файла, `parallel_plane_mono.ply`, который тоже можно скачать из репозитория книги на GitHub. Единственное различие между полигональной сеткой в этом примере и полигональной сеткой в PLY-файле `cube.ply` состоит в том, что теперь вместо шести сторон куба у нас только четыре грани, которые образуют две параллельные плоскости:

```
ply
format ascii 1.0
comment создан для книги Глубокое обучение в 3-D на Python
element vertex 8
property float32 x
property float32 y
property float32 z
element face 4
property list uint8 int32 vertex_indices
end_header
-1 -1 -1
1 -1 -1
1 1 -1
-1 1 -1
-1 -1 1
1 -1 1
1 1 1
-1 1 1
3 0 1 2
3 0 2 3
3 5 4 7
3 5 7 6
```

Полигональную сетку можно интерактивно визуализировать с помощью следующего ниже фрагмента исходного кода файла Python `ply_example2.py`.

1. Сначала импортируем все необходимые библиотеки Python:

```
import open3d
from pytorch3d.io import load_ply
```

[1] Также форма (англ. *shape*). – *Прим. перев.*

2. Затем, используя пакет Open3D, загружаем полигональную сетку:

```
mesh_file = 'parallel_plane_mono.ply'
print('Визуализация полигональной сетки с помощью Open3D')
mesh = open3d.io.read_triangle_mesh(mesh_file)
```

3. Далее применяем его метод `draw_geometries`, чтобы открыть окно для интерактивной визуализации полигональной сетки:

```
open3d.visualization.draw_geometries([mesh],
                                     mesh_show_wireframe=True,
                                     mesh_show_back_face=True)
```

4. Затем используем библиотеку PyTorch3D, чтобы открыть ту же полигональную сетку:

```
print('Загрузка того же файла с помощью PyTorch3D')
vertices, faces = load_ply(mesh_file)
```

5. И в конце распечатываем информацию о загруженных вершинах и гранях. На самом деле это просто обычные тензоры PyTorch3D:

```
print('Тип vertices = ', type(vertices),
      ', тип faces = ', type(faces))
print('vertices = ', vertices)
print('faces = ', faces)
```

Для каждой вершины также можно задать свойства, отличные от координат *x*, *y* и *z*. Например, можно задать цвета каждой вершины. Ниже приведен пример `parallel_plane_color.ply`:

```
ply
format ascii 1.0
comment создан для книги Глубокое обучение в 3-D на Python
element vertex 8
property float32 x
property float32 y
property float32 z
property uchar red
property uchar green
property uchar blue
element face 4
property list uint8 int32 vertex_indices
end_header
-1 -1 -1 255 0 0
1 -1 -1 255 0 0
1 1 -1 255 0 0
-1 1 -1 255 0 0
-1 -1 1 0 0 255
1 -1 1 0 0 255
1 1 1 0 0 255
-1 1 1 0 0 255
```

```
3 0 1 2
3 0 2 3
3 5 4 7
3 5 7 6
```

Обратите внимание, что в приведенном выше примере, наряду с *x*, *y* и *z*, мы также задаем несколько дополнительных свойств по каждой вершине, т. е. свойства `red`, `green` и `blue`, все это в типе данных `uchar`. Теперь каждая запись одной вершины представляет собой одну строку из шести чисел. Первые три числа – координаты *x*, *y* и *z*. Следующие три – значения RGB.

Полигональную сетку можно визуализировать с использованием файла Python `ply_example3.py` следующим образом:

```
import open3d
from pytorch3d.io import load_ply

mesh_file = 'parallel_plane_color.ply'
print('Визуализация полигональной сетки с помощью Open3D')

mesh = open3d.io.read_triangle_mesh(mesh_file)
open3d.visualization.draw_geometries([mesh],
                                     mesh_show_wireframe=True,
                                     mesh_show_back_face=True)
print('Загрузка того же файла с помощью PyTorch3D')
vertices, faces = load_ply(mesh_file)
print('Тип vertices = ', type(vertices),
      ', тип faces = ', type(faces))
print('vertices = ', vertices)
print('faces = ', faces)
```

В PLY-файле `cow.ply` мы также предоставляем реальный пример полигональной 3D-сетки. Читатели могут визуализировать указанную сетку с использованием файла Python `ply_example4.py`.

До сего момента мы говорили о базовых элементах формата файла 3D-данных PLY, таких как вершины и грани. Далее мы обсудим формат файла 3D-данных OBJ.

Формат файла 3D-данных – файлы OBJ

В этом разделе мы обсудим еще один широко используемый формат файла 3D-данных, формат файла OBJ. Формат файла OBJ был впервые разработан компанией Wavefront Technologies Inc. Как и формат файла PLY, формат OBJ также имеет как ASCII-версию, так и двоичную версию. Двоичная версия является проприетарной и незарегистрированной. И поэтому в данном разделе мы обсудим ASCII-версию.

Как и в предыдущем разделе, мы собираемся обследовать этот формат файла, посмотрев на примеры. Первый пример, `cube.obj`, приведен ниже. Как можно догадаться, в OBJ-файле определяется полигональная сетка куба.

```
mtllib ./cube.mtl
o cube
# Список вершин
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
v 0.5 -0.5 -0.5
v 0.5 0.5 -0.5
v 0.5 0.5 0.5

# Список точек/линий/граней
usemtl Door

f 1 2 3
f 6 5 8
f 7 3 2
f 4 8 5
f 8 4 3
f 6 2 1
f 1 3 4
f 6 8 7
f 7 2 6
f 4 5 1
f 8 3 7
f 6 1 5
```

Первая строка, `mtlib ./cube.mtl`, объявляет сопровождающий файл библиотеки шаблонов материалов (**MTL**)[1].

MTL-файл описывает свойства затенения поверхности, которые будут объяснены в следующем далее фрагменте исходного кода.

В строке `o cube` начальная буква `o` указывает на то, что в строке задается объект, имя которого – `cube`. Любая строка, начинающаяся с `#`, является строкой комментариев, т. е. остальная часть строки будет компьютером проигнорирована. В каждой строке, начинающейся с `v`, `v` указывает на то, что в данной строке задается вершина. Например, строка `v -0.5 -0.5 0.5` задает вершину с *x*-координатой `0.5`, *y*-координатой `0.5` и *z*-координатой `0.5`. В каждой строке, начинающейся с `f`, `f` указывает на то, что в данной строке содержится определение одной грани. Например, строка `f 1 2 3` задает грань, причем три ее вершины являются вершинами с индексами 1, 2 и 3.

Строка `usemtl Door` объявляет, что объявленные после этой строки поверхности должны быть затенены с помощью определенного в MTL-файле материального свойства под именем `Door`.

Сопровождающий MTL-файл `cube.mtl` приведен ниже. Указанный файл задает материальное свойство, именуемое `Door`:

```
newmtl Door
Ka  0.8 0.6 0.4
```

[1] От англ. *Material Template Library. – Прим. перев.*

```
Kd  0.8 0.6 0.4
Ks  0.9 0.9 0.9
d   1.0
Ns  0.0
illum 2
```

Мы не будем подробно обсуждать эти материальные свойства, кроме `map_Kd`. Если вам интересно, то можете обратиться к стандартному учебнику по компьютерной графике, такому как «Компьютерная графика: принципы и практика»[1]. Мы перечислим некоторые грубые описания этих свойств следующим образом, просто ради полноты:

- `Ka`: задает окружающий цвет,
- `Kd`: задает рассеянный цвет,
- `Ks`: задает бликовый цвет,
- `Ns`: определяет фокус бликовых моментов,
- `Ni`: определяет оптическую плотность (он же индекс рефракции),
- `d`: задает коэффициент растворения,
- `illum`: задает модель освещения,
- `map_Kd`: задает файл цветной текстуры, который будет применен к диффузной отражательной способности материала.

OBJ-файл `cube.obj` можно открыть как с помощью пакета Open3D, так и с помощью библиотеки PyTorch3D. Следующий ниже фрагмент исходного кода, содержащийся в файле Python `obj_example1.py`, можно скачать из репозитория книги на GitHub:

```
import open3d
from pytorch3d.io import load_obj

mesh_file = 'cube.obj'
print('Визуализация полигональной сетки с помощью Open3D')
mesh = open3d.io.read_triangle_mesh(mesh_file)
open3d.visualization.draw_geometries([mesh],
                                     mesh_show_wireframe=True,
                                     mesh_show_back_face=True)

print('Загрузка того же файла с помощью PyTorch3D')
vertices, faces, aux = load_obj(mesh_file)
print('Тип vertices = ', type(vertices))
print('Type faces = ', type(faces))
print('Тип aux = ', type(aux))
print('vertices = ', vertices)
print('faces = ', faces)
print('aux = ', aux)
```

В приведенном выше фрагменте исходного кода заданная полигональная сетка куба интерактивно визуализируется с помощью функции Open3D `draw_geometries`. Указанная сетка будет показана в окне, и вы сможете ее поворачивать, увеличивать и уменьшать ее масштаб, используя мышь. Полигональную

[1] От англ. *Computer Graphics: Principles and Practice. – Прим. перев.*

сетку также можно загрузить с помощью функции PyTorch3D `load_obj`. Функция `load_obj` вернет переменные `vertices`[1], `faces`[2] и `aux`[3] в формате тензора PyTorch либо в формате кортежей тензоров PyTorch.

Пример результата работы фрагмента исходного кода `obj_example1.py` приводится ниже:

```
Визуализация полигональной сетки с помощью Open3D
Загрузка того же файла с помощью PyTorch3D
Тип vertices = <class 'torch.Tensor'>
Тип faces = <class 'pytorch3d.io.obj_io.Faces'>
Тип aux = <class 'pytorch3d.io.obj_io.Properties'>
vertices = tensor([[-0.5000, -0.5000,  0.5000],
                   [-0.5000, -0.5000, -0.5000],
                   [-0.5000, 0.5000, -0.5000],
                   [-0.5000, 0.5000,  0.5000],
                   [ 0.5000, -0.5000,  0.5000],
                   [ 0.5000, -0.5000, -0.5000],
                   [ 0.5000, 0.5000, -0.5000],
                   [ 0.5000, 0.5000,  0.5000]])
faces = Faces(verts_idx=tensor([
                   [0, 1, 2],
                   [5, 4, 7],
                   [6, 2, 1],
                   ...
                   [3, 4, 0],
                   [7, 2, 6],
                   [5, 0, 4]]),
               normals_idx=tensor([
                   [-1, -1, -1],
                   [-1, -1, -1],
                   [-1, -1, -1],
                   [-1, -1, -1],
                   ...
                   [-1, -1, -1],
                   [-1, -1, -1]]),
               textures_idx=tensor([
                   [-1, -1, -1],
                   [-1, -1, -1],
                   [-1, -1, -1],
                   ...
                   [-1, -1, -1],
                   [-1, -1, -1]]),
               materials_idx=tensor([0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0]))
aux = Properties(
           normals=None,
           verts_uvs=None,
```

[1] Вершины. – *Прим. перев.*

[2] Грани. – *Прим. перев.*

[3] Вспомогательная переменная. – *Прим. перев.*

```
material_colors={
  'Door': {'ambient_color': tensor([0.8000, 0.6000, 0.4000]),
  'diffuse_color': tensor([0.8000, 0.6000, 0.4000]),
  'specular_color': tensor([0.9000, 0.9000, 0.9000]),
  'shininess': tensor([0.])}},
texture_images={},
texture_atlas=None)
```

Здесь из приведенной выше распечатки мы понимаем, что возвращенная переменная `vertices` представляет собой тензор PyTorch с очертанием 8×3, в котором каждая строка – это вершина с координатами *x*, *y* и *z*. Возвращенная переменная `faces` представляет собой именованный кортеж из трех тензоров PyTorch, `verts_idx`, `normals_idx` и `textures_idx`. В данном примере все тензоры `normals_idx` и `textures_idx` являются недопустимыми, поскольку файл `cube.obj` не содержит определения нормали и текстур. В следующем ниже примере мы увидим, как определять нормали и текстуры в формате OBJ-файла. `verts_idx` – это индексы вершин по каждой грани. Обратите внимание, что в PyTorch3D индексы вершин индексируются от нуля, т. е. отсчет индексов начинается с 0. Однако индексы вершин в OBJ-файлах индексируются от единицы, т. е. отсчет индексов начинается с 1. Библиотека PyTorch3D уже провела конверсию между двумя способами индексации вершины за нас.

Возвращаемая переменная `aux` содержит некоторую дополнительную информацию о полигональной сетке. Обратите внимание на пустое поле `texture_image`[1] переменной `aux`. Изображения текстур используются в MTL-файлах для задания цвета на вершинах и гранях. Опять же, мы покажем способы применения этой функциональности в следующем примере.

Во втором примере мы будем использовать пример из OBJ-файла `cube_texture.obj`, чтобы осветить некоторые другие функциональные возможности OBJ-файла. Ниже приведен сам файл:

```
mtllib cube_texture.mtl
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -0.999999
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vt 1.000000 0.333333
vt 1.000000 0.666667
vt 0.666667 0.666667
vt 0.666667 0.333333
vt 0.666667 0.000000
vt 0.000000 0.333333
vt 0.000000 0.000000
vt 0.333333 0.000000
vt 0.333333 1.000000
```

[1] Текстурное изображение. – *Прим. перев.*

```
vt 0.000000 1.000000
vt 0.000000 0.666667
vt 0.333333 0.333333
vt 0.333333 0.666667
vt 1.000000 0.000000
vn 0.000000 -1.000000 0.000000
vn 0.000000 1.000000 0.000000
vn 1.000000 0.000000 0.000000
vn -0.000000 0.000000 1.000000
vn -1.000000 -0.000000 -0.000000
vn 0.000000 0.000000 -1.000000
g main
usemtl Skin
s 1
f 2/1/1 3/2/1 4/3/1
f 8/1/2 7/4/2 6/5/2
f 5/6/3 6/7/3 2/8/3
f 6/8/4 7/5/4 3/4/4
f 3/9/5 7/10/5 8/11/5
f 1/12/6 4/13/6 8/11/6
f 1/4/1 2/1/1 4/3/1
f 5/14/2 8/1/2 6/5/2
f 1/12/3 5/6/3 2/8/3
f 2/12/4 6/8/4 3/4/4
f 4/13/5 3/9/5 8/11/5
f 5/6/6 1/12/6 8/11/6
```

Файл `cube_texture.obj` похож на файл `cube.obj`, за исключением следующих ниже различий:

- имеется несколько дополнительных строк, которые начинаются с `vt`. Каждая такая строка объявляет текстурную вершину с координатами `x` и `y`. В каждой текстурной вершине задается цвет, причем речь идет о пиксельном цвете на так называемом текстурном изображении, на котором местоположение пиксела – это *x*-координата ширины текстурной вершины *x* и *y*-координата высоты текстурной вершины *x*. Изображение текстуры будет определено в сопровождающем файле `cube_texture.mtl`;
- имеются дополнительные строки, которые начинаются с `vn`. В каждой такой строке объявляется вектор нормали – например, строка `vn 0.000000 -1.000000 0.000000` объявляет вектор нормали, указывающий на отрицательную ось *z*;
- каждая строка определения грани теперь содержит больше информации о каждой вершине. Например, строка `f 2/1/1 3/2/1 4/3/1` содержит определения трех вершин. Первая тройка, `2/1/1`, определяет первую вершину, вторая тройка, `3/2/1`, – вторую вершину, а третья тройка, `4/3/1`, – третью вершину. Каждая такая тройка – это индекс вершины, индекс текстурной вершины и индекс вектора нормали. Например, `2/1/1` определяет вершину, геометрическое местоположение которой определяется во второй строке, начинающейся с `v`, цвет определяется

в первой строке, начинающейся с vt, и вектор нормали определяется в первой строке, начинающейся с vn.

Сопровождающий файл cube_texture.mtl выглядит, как показано ниже. В нем начинающаяся с ключевого слова map_Kd строка объявляет текстурное изображение. Здесь wal67ar_small.jpg – это файл RGB-изображения 250×250 в той же папке, что и MTL-файл:

```
newmtl Skin
Ka 0.200000 0.200000 0.200000
Kd 0.827451 0.792157 0.772549
Ks 0.000000 0.000000 0.000000
Ns 0.000000
map_Kd ./wal67ar_small.jpg
```

Опять же, для загрузки находящейся в файле cube_texture.obj полигональной сетки можно использовать пакет Open3D и библиотеку PyTorch3D – например, используя следующий ниже файл obj_example2.py:

```
import open3d
from pytorch3d.io import load_obj
import torch

mesh_file = 'cube_texture.obj'

print('Визуализация полигональной сетки с помощью Open3D')
mesh = open3d.io.read_triangle_mesh(mesh_file)
open3d.visualization.draw_geometries([mesh],
                                     mesh_show_wireframe=True,
                                     mesh_show_back_face=True)
print('Загрузка того же файла с помощью PyTorch3D')
vertices, faces, aux = load_obj(mesh_file)
print('Тип vertices = ', type(vertices))
print('Тип faces = ', type(faces))
print('Тип aux = ', type(aux))
print('vertices = ', vertices)
print('faces = ', faces)
print('aux = ', aux)

texture_images = getattr(aux, 'texture_images')
print('texture_images type = ', type(texture_images))
print(texture_images['Skin'].shape)
```

Результат работы фрагмента исходного кода файла Python obj_example2.py должен выглядеть, как показано ниже:

```
Визуализация полигональной сетки с помощью Open3D
Загрузка того же файла с помощью PyTorch3D
Тип vertices = <class 'torch.Tensor'>
Тип faces = <class 'pytorch3d.io.obj_io.Faces'>
Тип aux = <class 'pytorch3d.io.obj_io.Properties'>
vertices = tensor([[ 1.0000, -1.0000, -1.0000],
                   [ 1.0000, -1.0000,  1.0000],
```

```
                  [-1.0000, -1.0000,  1.0000],
                  [-1.0000, -1.0000, -1.0000],
                  [ 1.0000, 1.0000, -1.0000],
                  [ 1.0000, 1.0000,  1.0000],
                  [-1.0000, 1.0000,  1.0000],
                  [-1.0000, 1.0000, -1.0000]])
faces = Faces(verts_idx=tensor([
                  [1, 2, 3],
                  [7, 6, 5],
                  [4, 5, 1],
                  [5, 6, 2],
                  [2, 6, 7],
                  [0, 3, 7],
                  [0, 1, 3],
                  ...
                  [3, 3, 3],
                  [4, 4, 4],
                  [5, 5, 5]]),
              textures_idx=tensor([
                  [ 0, 1, 2],
                  [ 0, 3, 4],
                  [ 5, 6, 7],
                  [ 7, 4, 3],
                  [ 8, 9, 10],
                  [11, 12, 10],
                  ...
                  [12, 8, 10],
                  [ 5, 11, 10]]),
              materials_idx=tensor([0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0]))
Aux = Properties(
           normals=tensor([[ 0., -1.,  0.],
                           [ 0., 1., 0.],
                           [ 1., 0., 0.],
                           [-0., 0., 1.],
                           [-1.,-0.,-0.],
                           [ 0., 0.,-1.]]),
           verts_uvs=tensor([[1.0000, 0.3333],
                             ...
                             [0.3333, 0.6667],
                             [1.0000, 0.0000]]),
           material_colors={
             'Skin': {'ambient_color': tensor([0.2000, 0.2000, 0.2000]),
             'diffuse_color': tensor([0.8275, 0.7922, 0.7725]),
             'specular_color': tensor([0., 0., 0.]),
             'shininess': tensor([0.])}},

          texture_images={'Skin': tensor([[[0.2078, 0.1765, 0.1020],
                                           [0.2039, 0.1725, 0.0980],
                                           [0.1961, 0.1647, 0.0902],
                                           ...,
                                           [0.2235, 0.1882, 0.1294]]])},
```

```
          texture_atlas=None)
texture_images type = <class 'dict'>
Skin
torch.Size([250, 250, 3])
```

Примечание

Это не полная распечатка; обратитесь к полной распечатке во время выполнения исходного кода.

По сравнению с распечаткой фрагмента исходного кода в файле `obj_example1.py` приведенная выше распечатка имеет следующие отличия.

- Поля `normals_idx` и `textures_idx` переменной `faces` теперь содержат валидные индексы вместо значения `-1`.
- Поле `normals` переменной `aux` теперь является тензором PyTorch, а не `None`.
- Поле `verts_uvs` переменной `aux` теперь является тензором PyTorch, а не `None`.
- Поле `texture_images` переменной `aux` больше не является пустым словарем. Словарь `texture_images` содержит одну запись с ключом `Skin` и тензором PyTorch с очертанием (250, 250, 3). Этот тензор точно такой же, как и содержащееся в файле `wal67ar_small.jpg` изображение, согласно определения в файле `mtl_texture.mtl`.

Мы научились использовать базовые форматы файлов 3D-данных, а также файлы PLY и OBJ. В следующем далее разделе вы изучите базовые понятия систем 3D-координат.

Понятие системы 3D-координат

В этом разделе вы познакомитесь с часто используемыми в библиотеке PyTorch3D системами координат. Данный раздел является адаптированной версией документации PyTorch по системам координат камеры: https://pytorch3d.org/docs/cameras. Для того чтобы понять и применять принятую в библиотеке PyTorch3D систему отрисовки, обычно необходимо знать эти системы координат и уметь их использовать. Как обсуждалось в предыдущих разделах, 3D-данные могут быть представлены точками, гранями и вокселами. Местоположение каждой точки может быть представлено набором координат *x*, *y* и *z* относительно определенной системы координат. Обычно требуется задать и использовать несколько систем координат, в зависимости от того, какая из них наиболее удобна.

Первая часто используемая система координат называется **мировой системой координат**. Эта система координат представляет собой систему 3D-координат, выбранную по отношению ко всем 3D-объектам, в результате чего местоположение 3D-объектов легко определяется. Обычно ось мировой системы координат не согласуется с ориентацией объекта или камеры. И поэтому между началом мировой системы координат и ориентациями объек-

та и камеры существуют ненулевые повороты и смещения. Ниже приведен рис. 1.3, показывающий мировую систему координат.

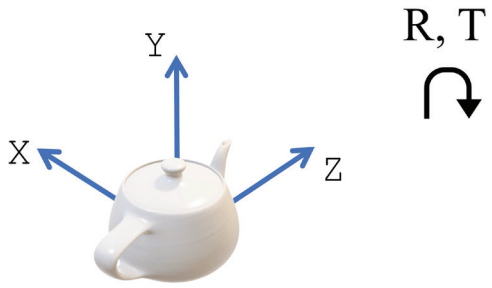

Рис. 1.2 ❖ Мировая система координат, начало координат и ось которой определяются независимо от позиций камеры

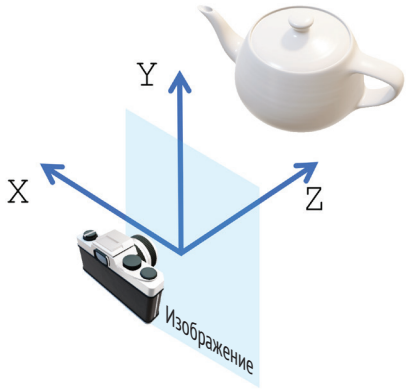

Рис. 1.3 ❖ Система координат поля зрения камеры, начало которой находится в центре проекции камеры, а три оси определяются в соответствии с плоскостью изображения

Поскольку ось мировой системы координат обычно не согласуется с ориентацией камеры, во многих ситуациях удобнее определять и использовать систему координат поля зрения камеры. В PyTorch3D система координат поля зрения камеры задается таким образом, что ее начало координат находится в точке проецирования камеры, ось *x* указывает влево, ось *y* указывает вверх, а ось *z* указывает вперед[1] (рис. 1.4).

Нормализованная координата устройства (NDC)[2] ограничивает объем, который камера может передавать. Значения координат *x* в пространстве NDC

[1] Иными словами, согласно документации PyTorch3D, система координат поля зрения камеры (Camera view coordinate system) – это система, начало которой находится в плоскости изображения, а ось *z* перпендикулярна плоскости изображения. – *Прим. перев.*

[2] От англ. *normalized device coordinate. – Прим. перев.*

варьируются в интервале от –1 до +1, как и значения координат *y*. Значения координат *z* варьируются в интервале от znear до zfar, где znear – это самая близкая глубина, а zfar – самая дальняя глубина. Любой объект вне этого диапазона между znear и zfar не будет передаваться камерой.

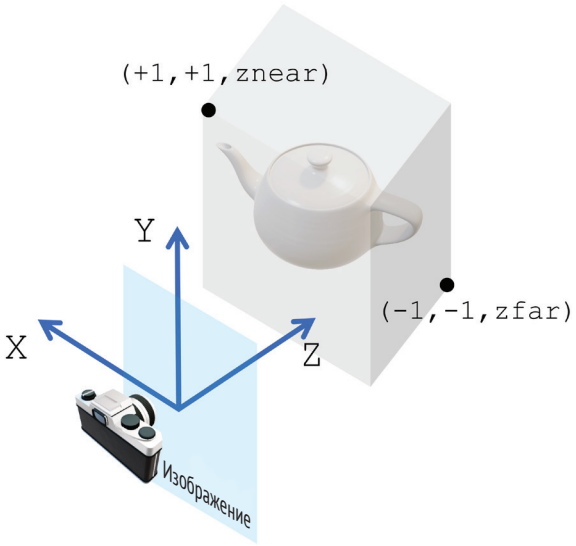

Рис. 1.4 ❖ Система координат NDC, где объем ограничен диапазонами, которые камера может передавать

Наконец, система координат экрана определяется с точки зрения того, как изображения отображаются на экранах. Данная система координат содержит координату *x* в виде столбцов пикселов, координату *y* в виде строк пикселов, а координата *z* соответствует глубине объекта.

В целях правильной отрисовки 3D-объекта на 2D-экране необходимо переключаться между этими системами координат. К счастью, такие конверсии легко выполняются с помощью применяемых в PyTorch3D моделей камеры. Мы обсудим конверсию координат подробнее после того, как рассмотрим модели камеры.

Понятие модели камеры

В этом разделе вы познакомитесь с моделями камеры. В трехмерном глубоком обучении 2D-изображения обычно используются для 3D-обнаружения. 3D-информация обнаруживается исключительно по 2D-изображениям, либо в целях получения высокой точности 2D-изображения смешиваются с глубиной. Тем не менее модели камеры необходимы для выстраивания соответствия между 2D-пространством и 3D-миром.

В библиотеке PyTorch3D есть две главнейшие модели камеры, ортографическая камера, определенная в классе `OrthographicCameras`, и перспективная камера, определенная в классе `PerspectiveCameras`. На следующем ниже рисунке показаны различия между этими двумя моделями камеры.

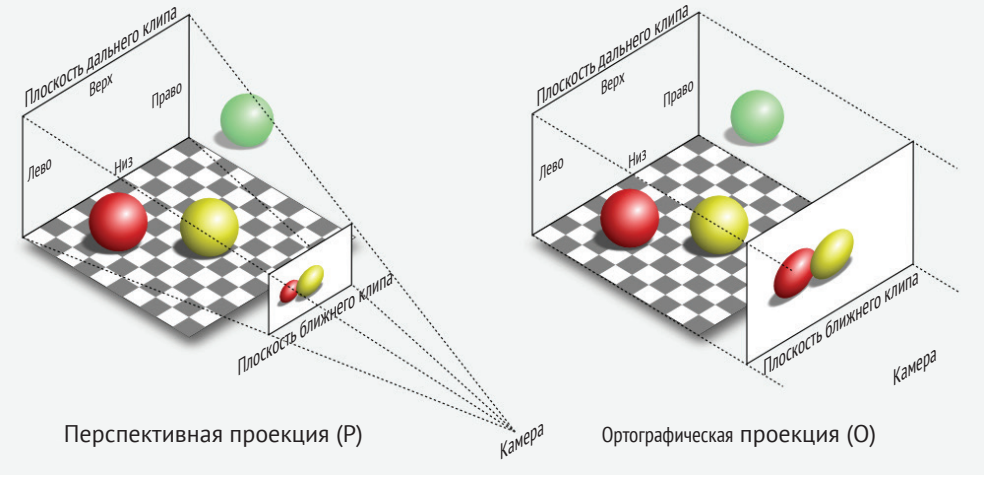

Рис. 1.5 ❖ Две реализованные в PyTorch3D главнейшие модели камеры: перспективная и ортографическая

В ортографических камерах используются ортографические проекции, чтобы отображать объекты 3D-мира на 2D-изображенияи, тогда как в перспективных камерах используются перспективные проекции, чтобы отображать объекты 3D-мира на 2D-изображения. Ортографические проекции отображают объекты на 2D-изображения, игнорируя глубину объекта. Например, как показано на рисунке, два объекта с одинаковым геометрическим размером на разных глубинах будут отображены на 2D-изображения одинакового размера. С другой стороны, в перспективных проекциях если объект отошел от камеры далеко, то на 2D-изображении он будет отображен на меньший размер.

Теперь, когда вы познакомились с базовым понятием моделей камеры, давайте рассмотрим несколько примеров программирования, чтобы понять, как создавать и использовать эти модели камеры.

Пример программирования моделей камеры и систем координат

В этом разделе мы применим все то, чему вы научились, чтобы разработать конкретную модель камеры и выполнять конверсию между разными системами координат, используя конкретный пример исходного кода, написанный на Python с использованием PyTorch3D.

1. Прежде всего будем использовать следующую ниже полигональную сетку, определенную в OBJ-файле `cube.obj`. По сути, полигональная сетка – это куб:

```
mtllib ./cube.mtl
o cube
# Список вершин
v -50 -50 20
v -50 -50 10
v -50 50 10
v -50 50 20
v 50 -50 20
v 50 -50 10
v 50 50 10
v 50 50 20

# Список точек/линий/граней
usemtl Door
f 1 2 3
f 6 5 8
f 7 3 2
f 4 8 5
f 8 4 3
f 6 2 1
f 1 3 4
f 6 8 7
f 7 2 6
f 4 5 1
f 8 3 7
f 6 1 5
# Конец файла
```

   Примера фрагмента исходного кода находится в файле Python `camera.py`, который можно скачать из репозитория книги на GitHub.
2. Далее импортируем все необходимые модули:

```
import open3d
import torch
import pytorch3d
from pytorch3d.io import load_obj
from scipy.spatial.transform import Rotation as Rotation

from pytorch3d.renderer.cameras import PerspectiveCameras
```

3. Загружаем и визуализируем полигональную сетку с использованием функции Open3D `draw_geometrics`:

```
# Загрузить полигональную сетку и визуализировать с помощью Open3D
mesh_file = "cube.obj»
print('Визуализация полигональной сетки с помощью Open3D')
mesh = open3d.io.read_triangle_mesh(mesh_file)
open3d.visualization.draw_geometries([mesh],
                                     mesh_show_wireframe=True,
                                     mesh_show_back_face=True)
```

4. Переменная `camera` определяется как объект PyTorch3D `PerspectiveCamera`. Камера здесь на самом деле мини-пакетирована. Например, матрица поворота, R, представляет собой тензор PyTorch с очертанием `[8, 3, 3]`, который фактически задает восемь камер, причем каждая с одной из восьми матриц поворота. Это касается всех других параметров камеры, таких как размеры изображений, фокусные расстояния и фокальные точки:

```
# Определить мини-пакет из 8 камер
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. После определения переменной `camera` можно вызвать метод `get_world_to_view_transform` класса, чтобы получить объект класса `Transform3D` под названием `world_to_view_transform`[1]. Затем можно применить метод `transform_points`, чтобы выполнить конвертацию из мировой системы координат в систему координат поля зрения камеры. Метод `get_full_projection_transform`[2] применяется точно так же, чтобы получить объект класса `Transform3D` под названием `world_to_screen_transform`[3], который

[1] Трансформанта мировой системы в систему координат обзора камеры. – *Прим. перев.*

[2] Получить трансформанту полной проекции. – *Прим. перев.*

[3] Трансформанта мировой системы в систему координат экрана. – *Прим. перев.*

предназначен для конвертации из мировой системы координат в систему координат экрана:

```
world_to_view_transform = camera.get_world_to_view_transform()
world_to_screen_transform = camera.get_full_projection_transform()

# Загрузить полигональные сетки с помощью 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)
```

В примере исходного кода показаны базовые способы применения камер библиотеки PyTorch3D и проиллюстрирована легкость, с которой можно переключаться между разными системами координат с использованием данной библиотеки.

Резюме

В этой главе вы сначала научились формировать среду разработки. Затем мы поговорили о наиболее широко используемых представлениях 3D-данных. Затем вы проинспектировали несколько конкретных примеров представления 3D-данных, изучив форматы файлов 3D-данных, формат PLY и формат OBJ. Затем вы познакомились с базовыми понятиями систем 3D-координат и моделей камеры. В последней части главы вы научились создавать модели камеры и конвертировать разные системы координат с помощью практического примера программирования.

В следующей главе мы поговорим о более важных понятиях трехмерного глубокого обучения, таких как отрисовка, чтобы конвертировать 3D-модели в 2D-изображения, разнородное мини-пакетирование и несколько способов представления поворотов.