МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Кафедра: Математического обеспечения и суперкомпьютерных технологий

Направление подготовки: «Прикладная математика и информатика»
Магистерская программа: «Вычислительные методы и суперкомпьютерные технологии»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Тема:
«Технологии VR/AR для 3D-реконструкции помещений: алгоритмы сканирования, обработки изображений и восстановления поверхностей на основе фотографических данных»

Выполнил:
студент группы 3823М1ПМвм

_____ Розанов Д.И.
Подпись

Научный руководитель:
Доцент кафедры математического обеспечения и суперкомпьютерных технологий, кандидат технических наук

_____Борисов Н.А.
Подпись

Нижний Новгород
2025

Содержание

Введение

Технологии виртуальной и дополненной реальности — это инновационные компьютерные технологии, позволяющие создавать виртуальные объекты и взаимодействовать с ними, а также с окружением в реальном времени. Технология виртуальной реальности (VR) — это технология, которая позволяет создавать полностью искусственное окружение, где пользователи могут перемещаться и взаимодействовать с объектами при помощи специальных устройств, например, VR-гарнитур. Дополненная реальность (AR), в отличие от виртуальной реальности, добавляет виртуальные объекты и информацию в реальный мир, используя камеры и дисплеи мобильных устройств, таких как смартфоны и планшеты.

Технологии виртуальной реальности имеют широкое применение в различных областях современного мира. Они используются для моделирования трехмерных пространств, визуализации объектов и создания более реалистичных изображений. Пользователи могут погрузиться в виртуальное пространство с помощью специальных устройств, что позволяет им более детально изучать объекты и взаимодействовать с ними, изменять их параметры и получить более глубокое понимание их формы и внешнего вида [1, 1].

На данный момент для получения 3D-моделей объектов, которые затем можно использовать в виртуальной реальности (VR), существуют несколько основных подходов с использованием фотографических данных: применение LIDAR-сканеров [2], 3D-сканеров [3] и фотограмметрия [4, 5]. Работа таких методов включает в себя: отслеживания объектов на изображениях по кадровой последовательности, либо построение карты глубины для их точек и оценивание расстояний. После чего, к полученному 3D-облаку точек применяются алгоритмы реконструкции поверхности, тем самым, на выходе получая аппроксимированную отсканированную модель с изображения.

Однако применение LIDAR-сканеров, 3D-сканеров и фотограмметрии обладает некоторыми недостатками. Все эти методы зависят непосредственно от освещения и условий съемки, что может привести к недостаточной точности и к потере качества данных при неоптимальных условиях [6]. Для LIDAR-сканеров и 3D-сканеров одним из недостатков является высокая стоимость оборудования и беспилотных систем для сканирования, что делает их недоступными для некоторых применений и ограничивает использование в широком масштабе. Также немаловажно, что они имеют ограничение по области обзора,

поэтому для сканирования крупных объектов требуется повторно менять положение сканера для полного охвата больших объектов или всей сцены. Все эти недостатки необходимо учитывать при выборе метода сканирования в зависимости от конкретных требований и условий работы. Из-за вышеперечисленных недостатков LIDAR-сканеров, 3D-сканеров и фотограмметрии, 3D-моделирование и реконструкция объектов (а также окружающего пространства) продолжает оставаться важной задачей в сфере визуализации.

В настоящее время существуют постоянные исследования и разработки новых методов и технологий, направленных на преодоление ограничений указанных методов. Например, последние исследования в области компьютерного зрения и машинного обучения позволили разработать алгоритмы, способные обрабатывать и анализировать большие объемы фотографических данных и достигать высокой точности и качества визуализации объектов. Так, к примеру, для 3D-сканирования объектов помещения, их реконструкции и последующего воссоздания в виртуальной реальности (VR) применяются следующие методы компьютерного зрения, как: методы основанные на анализе карты глубины (Depth Maps) [7], метод структуры движения (Structure from Motion) [8,10] и др.

Технологии виртуальной (VR) и дополненной реальности (AR) применяются в различных областях жизнедеятельности, и их использование в различных сферах растет с каждым годом:

1. Например, одной из возможностей использования виртуальной и дополненной реальности (VR/AR) является сфера образования. Благодаря этим технологиям, можно создавать интерактивные учебные материалы, которые позволят студентам усваивать знания и применять их на практике. Виртуальные сценарии (кейсы) помогают получить опыт в той или иной профессии и попробовать себя в какой-либо специальности;
2. Технологии VR/AR применяются также, например, в сфере архитектуры. Визуализация и реконструкция помещений, зданий, а также отдельных комнат позволяет более детально и подробно проработать требования к будущему проекту до этапа конструирования. Более того, при работе над крупными проектами исправление ошибки в требованиях, обнаруженной на этапе проектирования обходятся «дороже», чем исправление аналогичной ошибки на этапе выработки требований, что позволяет значительно снизить расходы на реализацию проекта по строительству какого-либо архитектурного объекта.

VR-технология все чаще применяется в сферах, где необходимо удобство и быстрота в использовании виртуальных 3D-моделей, полученных сканированием реальных помещений.

В данной работе будет рассмотрена тема визуализации и реконструкции помещений на основе фотографических данных при использовании VR/ AR технологий. Так же для этого будут рассмотрены и применены некоторые методы для обработки фотографий, методы из компьютерного зрения и методы машинного обучения.

На настоящий момент существует большое количество инструментов для воссоздания 3D-моделей из видео или набора фотографий, и большинство из них для корректной работы требует от пользователя точную информацию о модели камеры и ее внутренние параметры (фокусное расстояние, разрешение, матрица камеры). Но не всегда для моделирования объектов известны характеристики камеры в случае, если выборка фотографий приведена из интернета. Как отмечают в своей работе Даминов И. А. и соавторы (Арсенюк А. Ю., Тощев А. С.): «Трехмерная (3D) реконструкция — важный раздел компьютерного зрения и фотограмметрии, позволяющий создавать точные и реалистичные цифровые представления объектов или сцен из двумерных изображений или облаков точек. Однако существующие методы 3D-реконструкции часто сталкиваются с проблемами, связанными с пространственной точностью и визуальным реализмом» (Даминов И. А. Комбинирование алгоритмов SfM и ORB при 3D-реконструкции – 2023, с. 456 – 465). Так, например, для визуализации и реконструкции 3D-моделей используют следующие методы:

1. метод `bundle adjustment` или метод `bundler`, который является системой структуры слежения за движением (`SfM - Structure-from-Motion`) для неупорядоченных коллекций изображений, написанная на языке C/ C++. `Bundler` в качестве входных данных принимает набор изображений, характеристик и совпадений изображений, а на выходе создает трехмерную реконструкцию геометрии камеры - система реконструирует сцену постепенно, по несколько изображений за итерацию.
2. методы OpenCV (open source computer vision library) – методы из библиотеки алгоритмов компьютерного зрения, обработки изображений и численных алгоритмов общего назначения. Например, функция `reprojectImageTo3D`, на вход которой поступает карта смещения, полученная от двух ректифицированных фотографий, снятых на откалиброванную камеру, а на выходе функция возвращает трехмерный результат смоделированного изображения.

Но так как не всегда есть необходимость получить реальные размеры предметов, то используют и другие методы для моделирования и реконструкции. Например, метод оптического потока, который отображает видимое движение объектов сцены относительно наблюдателя на визуальном графике или в виде математической модели.

Выбранная мной тема актуальная, так как задача визуализации и реконструкции помещений на основе фотографических данных позволяет смоделировать точную и детальную модель помещения, которая может быть использована в различных сферах жизни. Так, тема использования технологий VR/AR для визуализации и реконструкции помещений на основе фотографических данных важна и актуальна для дальнейшего развития такой сферы, как «криминалистика». Этот проект в дальнейшем будет использован и направлен на создание точной и реалистичной виртуальной модели места преступления, основанной на доступных фотографических данных. Используя современные технологии VR/AR, криминалисты смогут воссоздать трехмерную среду, сильно приближенную к реальной, исследуя ее с разных углов и перспектив. Этот же инструмент предоставит криминалистам возможность более детально изучить место преступления без физического присутствия на месте происшествия. Они смогут взаимодействовать с виртуальной моделью, перемещаться по ней, изменять ракурс обзора и увеличивать детали. Такая возможность позволяет идентифицировать потенциальные улики и анализировать другие факторы окружения, которые могут быть связаны с преступлением, например: дождливая или ветренная погода, освещение на месте преступления и т.д.

Объектом исследования данной курсовой работы являются технологии VR/AR для создания точной модели помещений на основе фотографических данных, и выбранная мной тема позволяет исследовать возможности применения современных технологий VR/AR. Это может также и иметь практическое применение в различных областях сферы жизни. Кроме того, исследование выбранной темы может в дальнейшем привести к разработке новых методов и алгоритмов для обработки фотографических данных и создания более качественных визуализаций помещений.

Предмет исследования – методы и алгоритмы, которые используются вместе с технологиями виртуальной и дополненной реальности для создания трехмерных моделей помещения на основе фотографических данных, а также их применение в различных областях.

Глава 1. Предмет исследования

1.1. Основные определения

Виртуальная реальность (virtual reality, VR) - мир, созданный при помощи технических средств, меняющий восприятие окружающей среды. Поведение объектов приближенно к их поведению в реальном мире. Пользователь может воздействовать с этими объектами в соответствии с реальными законами физики, например, таких как, гравитация, текучесть воды, столкновение с предметами, отражение.

Дополненная реальность (Augmented reality, AR) – это технология, которая с помощью устройств (таких как смартфоны, планшеты или специальные очки) позволяет создавать впечатление того, что виртуальные элементы (изображения, звуки, тексты) существуют (добавлены) в реальном мире.

XR-технологии (расширенная реальность) – обобщающее понятие для всех иммерсионных форматов, а именно: виртуальная реальность (VR), дополненная реальность (AR), смешанная реальность (MR)

1.2. Инструменты и программные решения для управления XR- проектами

В качестве среды для 3D моделирования в дальнейшем будем использовать ПО «Blender». Blender позволяет визуализировать сцены и модели, построенные и записанные в определённом формате для загрузки. Проект будет разработан с использованием игрового движка Unity версии v 2021.3.29f1 (либо любая версия выше 3.29).

1.3. Постановка задачи

Основной целью данной работы является реализация программного обеспечения способного в VR реконструировать комнату, используя фотографические данные. В связи с поставленной целью исследования для ее достижения необходимо выполнить следующие задачи на данный семестр:

1. Ознакомиться с научными материалами и изучение литературы на данную тему;
2. проанализировать имеющуюся информацию и изучить методы решения поставленной задачи в рамках данной темы;
   2.1. изучить технологии AR для реконструкции помещений: PlaneTracking (отслеживание вертикальных/ горизонтальных плоскостей)
   2.2. изучить методы распознавания объектов (`stereoRectify + stereoBM` и метод "плотного" потока)
3. спроектировать программный комплекс;
4. визуализация комнаты в VR (реконструкция помещения и обработка данных): проектирование программного комплекса, реализация изученных методов в п.2-3, частичная реставрация комнаты в 3D Unity (стены, пол и потолок с использованием серии фотографии, сделанных из центра комнаты)
5. проанализировать полученные результаты и сделать выводы - определить дальнейшие перспективы для изучения темы исследования и перспективы для применения полученных результатов;

Глава 2. Настройка Unity

2.1. Настройка Unity для работы с AR

Установка плагинов

Для работы с AR необходимо установить следующие плагины в проект:

1. `AR Foundation` – «обертка» над низкоуровневыми интерфейсами программы (например: `ARKit` и `ARCore`). Позволяет создавать приложения, которые могут работать на различных устройствах и ОС. Облегчает процесс разработки, предоставляя разработчикам доступ к общим функциям AR, таким как: распознавание объектов, отслеживание движения и визуализация AR-элементов.
2. `ARCore XR plugin` – плагин, который позволяет создавать приложения с `AR` на базе `ARCore`, и предоставляет доступ для работы приложения на ОС Android

Настройка сцены в Unity

Основная сцена: main

1. Необходимо удалить «`Main Camera`» (созданная автоматически при создании новой сцены), т.к. в AR-приложениях камера используется для захвата изображения с камеры устройства, а не для отображения игровой сцены.
2. Создаем объект XR «`AR Session`»
3. Создаем объект XR «`AR Session Origin`»
4. Устанавливаем «`AR Session`» и «`AR Session Origin`» по умолчанию в нулевую позицию $(x, y, z = 0)$
5. Для объекта «`AR Session Origin`» необходимо у его “сына «`AR Camera`» изменить параметр «`Cleaping place`» на 0.01, т.к. масштаб сцены в unity 1:1 (т.е., например, куб $1 \times 1 \times 1 = 1$ м^3)
6. Для сборки проекта под ОС Android: необходимо в «`Build setting`» сменить платформу с Windows под `Android (Switch platform)`

Настройка параметров и свойств проекта (Edit: Project settings)

1. Player (настройки платформы, управления памятью и сборки):
   1.1. Убрать галочку «Auto Graphics API»
   1.2. Убрать «Vulcan», т.к. ARCore работает только с OpenGL, а с Vulcan работать не умеет
   1.3. Для настройки гамма-коррекции необходимо перейти в меню Rendering/ Color Space и выбрать параметр Gamma (отвечает за то, как Unity обрабатывает цвета в проекте, и gamma-коррекция используется для компенсации нелинейности человеческого зрения и обеспечения более точного отображения цветов на экране)
   1.4. Необходимо выставить минимальную версию android v. 7.0., т.к. обязательно наличие ARCore на устройстве, иначе при сборке проекта будет следующая ошибка: «Android 7.0 'Nougat'»
2. XR Plug-in Management (управление подключениями XR-плагинов)
   2.1. Необходимо указать, что запускаем XR на старте приложения на ARCore
   2.2. В разделе ARCore в «Requirement» поставить вместо required – optimal, т.е. указать, что у телефона с ОС Android должен быть встроен ARCore. В некоторых же случаях можно запустить AR приложение на Android без ARCore, но с настройкой optimal количество устройств, которое поддерживает приложение – больше
   2.3. Depth нужно с required заменить на optimal, т.к. телефон может не поддерживать Depth required, и приложение может не запуститься при сборке.

Теперь, после установки соответствующих настроек, можно запустить приложение на устройстве, поддерживающем дополненную реальность (AR), и после запуска приложения на таком устройстве можно использовать возможностями дополненной реальности.

2.2. Настройка Unity для работы с VR

Настройка параметров и свойств проекта (Edit: Project settings)

XR Plug-in Management (управление подключениями XR-плагинов):

1. Необходимо установить XR Plugin: «Install XR Plugin Management»
2. В «Plug-in Provides» необходимо выбрать «OpenXR», т.к. данная технология виртуальной реальности поддерживается у многих шлемов, и устанавливается дополнительная установка пакетов.
3. В разделе OpenXR/ Interactions Profiles необходимо указать профили взаимодействия. Поставим все возможные профили взаимодействия контроллеров, т.к. таким образом, приложение будет более кроссплатформенным: Eye Gaze Interaction Profile, HTC Vive Controller Profile, Khronos Simple Controller Profile, Microsoft Hand Interaction Profile, Microsoft Motion Controller Profile, Oculus Touch Controller Profile, Valve Index Controller Profile. Но будем использовать только Oculus Touch Controller Profile.

Управление и установка пакетов и расширений (Package manager)

Для работы с VR необходимо установить следующие расширения в проект:

1. Необходимо добавить toolkit «com.unity.xr.interaction.toolkit» и заимпортировать «Starter assets», «XR Device Simulator», «Tunneling Vignette».
2. Добавим в проект пакет «OculusHands.unitypackage» для отображения рук.

Настройка сцены

Основная сцена: main

1. Необходимо удалить «Main Camera» (созданная автоматически при создании новой сцены), т.к. для виртуальной реальности используется специальная камера, которая уже встроена в систему VR. Удаление основной камеры помогает избежать конфликтов между двумя камерами и обеспечивает правильную работу VR-интерактивности.

2. Создаем объект XR Origin VR - начало координат для позиции и ориентации игрока в виртуальной реальности. Объект включает в себя камеру «Camera Offset», который является точкой отсчета для камеры, и еще три объекта: «Main Camera», «LeftHand Controller», «RightHand Controller». Камера отображает виртуальное окружение, а контроллеры для левой и правой руки позволяют пользователю взаимодействовать с виртуальным миром.
   2.1. Расположим XR Origin/ Camera Offset в position (0f, 0f, 0f).
   2.2. На объект XR Origin VR добавим встроенный скрипт Locomotion System (для правильной организации движения нашего пользователя) и в качестве компонент укажем наш объект XR Origin.
   2.3. В объект XR Origin добавим объект XR Interaction Manager для взаимодействия с окружением.
   2.4. Для LeftHand Controller и RightHand Controller уже прописан и добавлен в качестве компонент скрипт «XR Controller» и скрипт «XR Ray Interactor». Скрипт «XR Controller» отвечает за управление контроллерами виртуальной реальности, и обычно привязан к объектам, которые представляют контроллеры. А скрипт «XR Ray Interactor» отвечает за обнаружение и взаимодействие с объектами в VR при помощи лучей raycasting, которые позволяет определить, на что указывает позиция и направление контроллера в пространстве VR. Когда луч сталкивается с объектом или поверхностью, XR Ray Interactor может обрабатывать различные события. Для использования скрипта «XR Controller» требуется настроить параметры для каждого контроллера - нужно выбрать соответствующие настройки для левого и правого контроллеров: «XRI Default Left Controller» и «XRI Default Right Controller», что позволит корректно связать функциональность каждого контроллера с другими скриптами и обеспечить правильное взаимодействие виртуальной реальности.
   2.5. Добавим из пакета «OculusHands.unitypackage» модели рук для левого и правого контроллера

Настройка передвижения

Параметры и компоненты объекта XR Origin:

1. Добавим компоненту Character Collider, чтобы пользователь мог двигаться по плоскости. В качестве настроек Character Collider необходимо: уменьшить radius до 0.1 (т.к. иначе будут коллизии и столкновения с другими объектами), изменить height на 1.65 (~росту 1.65м), ограничить угол поверхности, на которую пользователь может подняться: slope limit 45 и ограничить высоту ступеньки, на которую пользователь может подняться step offset.
2. Добавим встроенный скрипт «Continuous Move Provider (Action-based)» и в качестве системы движения нужно указать Locomotion system.

   Также нужно установить значение «Enable strafe» и «Enable Gravity» true, чтобы использовать гравитацию и передвигаться влево/вправо не поворачиваясь камерой, а «Gravity Application Mode» нужно поставить в Immediately, чтобы движение было настроено по джойстику или по направлению камеры.

   В качестве Forward source нужно указать направление, по которому двигаемся – это вектор (нормаль), поэтому можем прикрепить в качестве этого параметра объект «Main Camera». В «Right Hand Move Action» необходимо указать конкретный объект, при взаимодействии с которым будет происходить движение: возьмем его из toolkit XRI LeftHand Locomotion / Move, и укажем, что при взаимодействии с правым джойстиком будет происходить движение персонажа в виртуальной среде.
3. Добавим компоненту «Continuous Turn Provider», которая отвечает за плавный поворот.

   В System прикрепляем Locomotion System, и укажем ссылку на левую руку, выбрав XRLocomotion Turn, чтобы при взаимодействии с левым контроллером происходил поворот камеры вокруг оси персонажа.

Теперь приложение виртуальной реальности (VR) поддерживает настройку и расширение функционала. Реализована система передвижения с помощью контроллеров, а также возможность вращения головой и осмотра всего вокруг с помощью шлема VR

Глава 3. Визуализация комнаты в VR: Реконструкция и обработка данных (анализ фотографии из центра комнаты)

3.1. Описание идеи

На вход поступает как минимум одна фотография из центра комнаты и текстовый файл с двумерным массивом расстояний до объектов. Необходимо построить комнату с правильным расположением объектов, используя доступные данные.

Для построения стен, пола и потолка комнаты на основе фотографии из центра комнаты, необходимо:

1. найти информацию о стенах в текстовом файле. Они обозначены как «wall1», «wall2», «wall3» и «wall4»;
2. среди них найти стену, расположенную напротив камеры, и получить расстояние до нее.
3. построить противоположную стену, отражая ее в обратную сторону относительно камеры, при этом сохраняя такое же расстояние.
4. разместить боковые стены на расстоянии, равном половине длины выбранной стены, как справа, так и слева
5. загрузить размеры стены из файла и отредактировать их с помощью соответствующей функции
6. построить потолок и пол на основе полученных размеров, создав плоскость с длиной, равной двукратной расстоянию до стены перед камерой, и шириной, равной длине выбранной стены.

Таким образом, используя указанные данные и операции, можно легко построить нужные элементы комнаты.

3.2. Методы решения

Для загрузки изображений из папки, добавлена кнопка «`Load Image`», и прописан соответствующий скрипт. Фотографии должны быть в формате `.jpg`. Считываемые изображения находятся в папке «Image», расположенной по пути «`/Assets/App/Image`». Так же реализована кнопка «Save Image», при нажатии которой обработанные фотографии сохраняются в корневую папку, а по кнопке «Clear» можно удалить загруженные в память изображения.

Написан отдельный скрипт для кнопки «Add obj» для добавления объектов на сцену. По нажатии кнопки, скрипт создаст новые объекты, считанные с фотографии с описанием из текстового файла, и разместит его на нужном расстоянии от камеры. Стены, пол и потолок добавляются, как было описано ранее в п. [6.1.1. Описание идеи].

3.3. Программная реализация

Программная реализация выполнена на языке C# в среде Microsoft Visual Studio 2022. Полная реализация перечисленных ниже методов и классов представлена в [Приложение D].

Используемые типы данных

Для пользовательских типов данных было создано пространство имен «App.Code», и вся программная реализация находится в данном namespace.

1. Struct Wall – структура для объекта «wall», описывающая положение стены и ее размер.
2. Class AddObject – класс, содержащий в себе список добавленных предметов, список стен, и игровые объекты (prefabs) для конкретных объектов, добавляемых на сцену.
3. GameObject – игровой объект: для создания и управления объектами в сцене игры.

Описание структуры Wall:

Поля структуры:

1. pos_x, pos_y, pos_z – позиции стены в игровом пространстве
2. scale_x, scale_y, scale_z – масштаб стены в игровом пространстве
3. distance – расстояние до стены от камеры.

Описание класса AddObject

Поля структуры:

1. object_list – список игровых объектов, добавленных на сцену
2. walls – список стен, добавленных на сцену
3. globalScale – параметр масштаба сцены.

Методы класса:

1. `Wall GetWall(string[])` – метод для получения описания параметров стен из текстового файла. На вход поступает строка, которая начинается с ключа «`wall`», и считываются остальные данные для этой строки. На выходе возвращается объект типа `Wall` с параметрами, соответствующими считанным из файла.
2. `void AddObject(GameObject, Vector3, Quaternion, scale)` – метод для добавления объекта типа данных `GameObject` на сцену в позицию, заданную Vector3, под углом, соответствующему `Quaternion` и размера scale
3. `void AddWall()` – метод, открывающий текстовый файл с описанием объектов. Метод проходит по строкам текстового файла и ищет ключ «`wall`». Затем для считанной строки вызывается описанный выше метод `AddObject`. Длина, ширина, местоположение стены в игровом пространстве определяется соответственно п. [Описание идеи].
4. `void AddFloor()` и `void AddRoof()` – методы, добавляющие пол и потолок соответственно.

3.4. Результаты

Запустив программную реализацию на примере комнаты 3×5 м. с потолками высотой 4 м. визуализируется следующая комната (рисунок. 1):

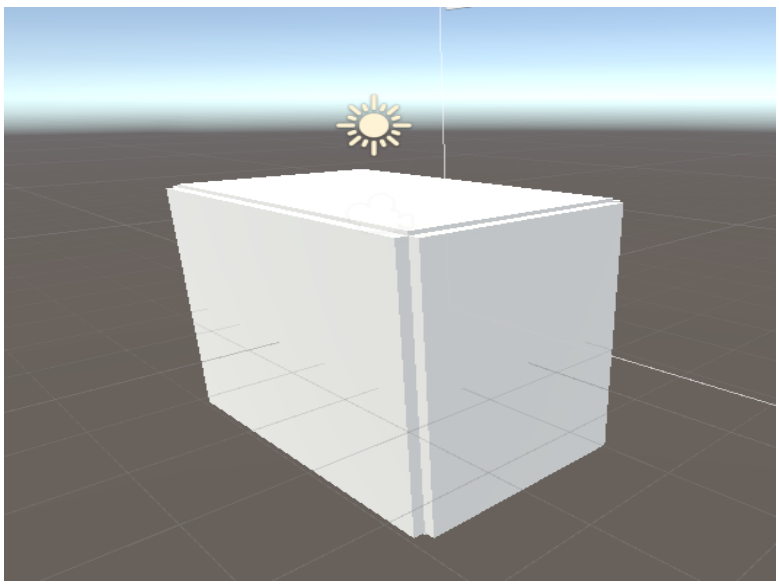


Рисунок. 1. визуализация комнаты (стены, пол, потолок).

Камера (игрок) находится в центре комнаты, в позиции (0f, 0f, 0f). До стены напротив игрока должно быть 1.5м (т.к. комната в ширину 3м), т.е. позиция противоположной стены (0f, 0f, 1.5f) (рисунок. 2):

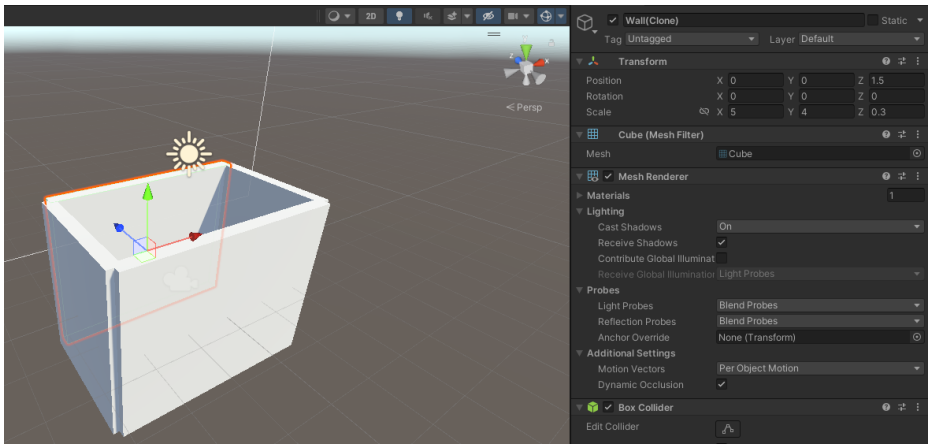


Рисунок. 2. проверка на совпадение описания стены.

Таким образом, стены находятся в позициях (0f, 0f, -1.5f), (0f, 0f, 1.5f), (-2.5f, 0f, 0f), (2.5f, 0f, 0f). Потолок и пол в позиции соответственно (0, 2, 0) и (0, -2, 0). Данные из файла о конструкции комнаты считаны корректно, и на этих данных построены стены, пол и потолок в правильных позициях.

Глава 4. Поиск вертикальных и горизонтальных плоскостей в AR Unity (PlaneTracking)

4.1. Описание идеи

Изначально иерархия объектов сцены выглядит следующим образом:

1. Сцена: Main
    1.1. Свет сцены: Directional Light
    1.2. XR объект: AR Session Origin
        1.2.1. Камера: AR Camera
    1.3. XR объект: AR Session

К объекту AR Session Origin необходимо добавить компоненту: объект «AR Plane Manager» и в «Detection Mode» указать, что нужно отслеживать и горизонтальные поверхности, и вертикальные (everything).

Чтобы данные плоскости визуализировались – создадим плоскость, на нее добавим Mesh Render, скрипт AR Plane и AR Plane Mesh Visualizer (script), а также наложим на нее белый материал с прозрачностью. Сделаем из этой плоскости prefab и в качестве компоненты Plane Prefab для объекта AR Plane Manager прикрепим этот prefab для плоскости.

Если запустим собранную программу на телефоне, то мы увидим, что отображаются плоскости, найденные программой. ARCore пытается распознать все плоскости, но не всегда это хорошо получается (такие особенности у ARCore). Иногда появляются плоскости, которых нет или не до конца прорисованные плоскости, или нет плоскостей, которые на самом деле существуют. В таком случае в программных реализациях используют, например, датчик глубины, чтобы понять, что какой-то из плоскостей нет. Но ARCore такое исправить не сможет. Таким образом, мы научились распознавать плоскость, но теперь нужно научиться распознавать тип плоскости: вертикальный или горизонтальный. Для этого в коде необходим проверить alignment plane. Например, мы нажимаем на экране телефона на область, где визуализирована какая-то плоскость. Мы проверяем, что точно было нажатие на плоскость или на какой-то же другой объект. Если было касание и пересечение Raycast является плоскостью, то мы можем получить у выбранной плоскости компоненту alignment и

сравнить ее с параметром `PlaneAlignment.Vertical`. Если сравнение вернет true, то плоскость является вертикальной, а иначе горизонтальной.

Так, для проверки, что плоскости правильно определяются на свойство вертикальности/ горизонтальности: пусть будем при нажатии на экране на выбранную плоскость добавлять объект куб размера $0.1 \times 0.1 \times 0.1$ или же сферу такого же размера в случае, если соответственно нажали на вертикальную плоскость или на горизонтальную. Добавление сферы и куба будет ставиться только на нажатие. Добавим сервис (пустой объект), и в него в качестве компоненты добавим написанный скрипт, указав префабы для двух объектов: для куба и сферы, созданные заранее.

Теперь же если запустить собранное приложение, то мы увидим все плоскости, которые `ARCore` смог распознать. Визуально можем сами же определить, какие из них вертикальные или горизонтальные, но также, коснувшись какой-то из плоскостей – на ней появится куб или же сфера в зависимости от ее типа расположения.

4.2. Методы решения

Для поиска вертикальных и горизонтальных поверхностей в AR был написан скрипт «`ARPlaceObject.cs`». Внутри в пространстве имен реализован класс с методом Update, внутри которого реализован код, определяющий тип поверхности, а также тестирование описанной выше программной реализации (добавление куба или сферу размера $0.1 \times 0.1 \times 0.1$ на горизонтальную или вертикальную плоскость).

Чтобы определить, какой тип у плоскости, для начала необходимо проверить, что нажатие было именно на плоскость: сравнивается, что свойство `trackable` для текущего объекта `hits[0]` из списка `List<ARRaycastHit>` сравнимо с объектом `ARPlane`. Свойство `trackable` используются для связи с обнаруженными объектами в реальном мире, например, с такими как: плоскости `(ARPlane)`, точки `(ARPoint)` или другие AR-объекты. Затем, в ветке `true` для объекта `ARPlane`, выполняется проверка, является ли плоскость вертикальной. После, для дальнейшего тестирования, как было описано раннее, создается `prefab` либо куба, либо сферы. Вызывается метод `SpawnPrefab`, который создает `prefab` в указанной точке, и в зависимости от результата последнего сравнения на тип плоскости, добавляется `prefab` либо куба, либо сферы соответственно. На вход метода `SpawnPrefab`

подается объект типа данных GameObject, который и не обходимо создать на выбранной плоскости, а на выход функция ничего не возвращается (тип данных void).

Для обработки нажатия на плоскость для начала необходимо было проверить, что свойство Input.touchCount не сравнимо с нулем, иначе касания по экрану не было и происходит выход из функции. Затем, если все же касание было, то проверяется, что фаза первого касания Input.GetTouch(0) сравнима с фазой TouchPhase.End - т.е. касание, которое было активно на экране, завершилось. Затем у этого же касания можем получить позицию, и передать ее в качестве аргумента в функцию Raycast, которая используется для выпуска луча из указанной точки на экране и определения объектов или поверхностей, с которыми луч пересекается.

На вход для функции Raycast передается позиция, откуда выпущен луч GetTouch(position), массив структур данных для хранения результатов пересечения луча с объектами или поверхностями List<ARRaycastHit> hits и PlaneWithinBounds - тип отслеживаемого объекта или поверхности, с которыми выполняется пересечение с лучом (PlaneWithinBounds означает, что мы ищем только поверхности, которые находятся в пределах границы трекинга (boundaries)). После выполнения метода RayCastManager.Raycast(), результат пересечения луча с объектами или поверхностями будет сохранен в переменную result, а информация о пересечении будет доступна в списке hits. Эта же информация используется для дальнейшей обработки, чтобы разместить объект на найденной поверхности в соответствии с результатами пересечения луча и поверхности. Так, затем проверяется, что результат не пустой, и далее идет проверка на тип плоскости.

4.3. Описание программной реализации

Программная реализация planeTracking выполнена на языке C# в среде Microsoft Visual Studio 2022. Полная реализация перечисленных ниже методов и классов представлена в [Приложение А. PlaneTracking].

Используемые типы данных

1. ARPlaceObject – класс, отвечающий за отслеживанием плоскости в дополненной реальности и размещение объектов на основе касания пользовательского ввода. Для пользовательских типов данных было создано пространство имен App.Code, и класс находится в этом namespace
2. ARRaycastManager – используется для raycasting`a
3. GameObject – игровой объект: для создания и управления объектами в сцене игры.
4. List<ARRaycastHit> - список пересечений луча отслеживания виртуального объекта с реальными объектами в окружающей среде
5. List<GameObject> - список для игровых объектов, добавленных на сцену

Описание класса ARPlaceObject

Поля класса:

1. CubeToPlace и SphereToPlace – ссылки на куб и сферу для добавлений их на сцену
2. hits – список с информацией о пересечении луча с объектами
3. instances – список добавленных объектов на сцену

Методы класса:

1. void Awake()- метод, вызывающийся один раз при инициализации объекта. Используется для инициализации переменных, настройки компонент и установки начальных значений - создает два новых списка объектов типа List<GameObject> и List<ARRaycastHit>
2. void SpawnPrefab GameObject obj) – создает на сцене объект obj с указанной позицией и вращением, полученными из списка hits[0].pose, и добавляет его в список добавленных объектов на сцену (instances)
3. void Update() – вызывается обновление каждого кадра, а внутри происходит проверка на тип плоскости и добавление объекта на сцену.

4.4. Результаты

Запустим программную реализацию на ОС Android и протестируем. По результатам проверки, видим, что успешно находятся горизонтальные и вертикальные плоскости комнаты. Как и было описано в функционале программы, на горизонтальные плоскости добавляются сферы, а на вертикальные кубы (рисунок 3-4):

Рисунок. 3. Пример: горизонтальные поверхности

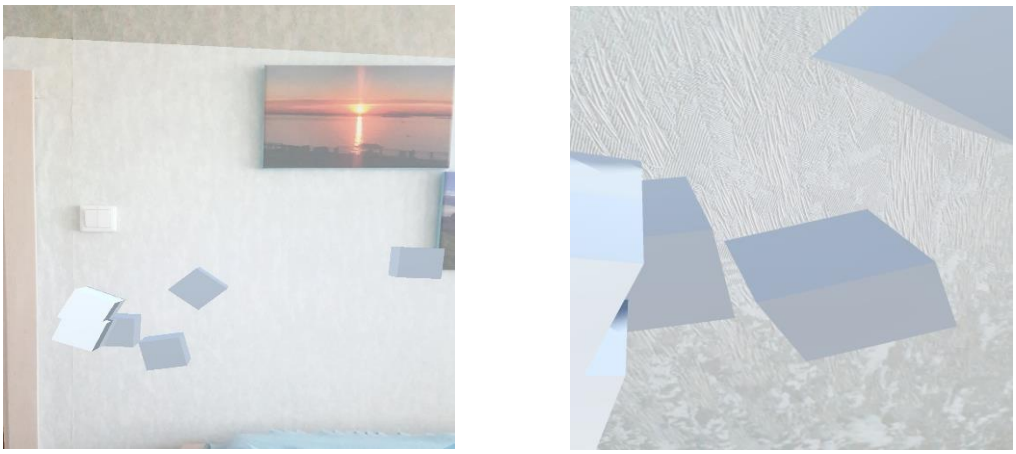


Рисунок. 4. Пример: вертикальные поверхности

Программная реализация работает корректно и правильно распознает вертикальные и горизонтальные плоскости, используя технологии дополненной реальности AR.

Глава 5. Методы распознавания объектов: плотный поток Фарнебака и стереоректификация с использованием stereoBM

5.1. Постановка задачи

Основной целью данного раздела является реализация одного из методов оптического потока и метода, основанного на стереоректификации и `stereoBM`, для задачи распознавания объектов с использованием в качестве входных данных двух последовательных фотографий. Данная цель предполагает решение следующих основных задач:

1. Изучение и анализ методов оптического потока.
2. Изучение и анализ метода, основанного на стереоректификации и stereoBM.
3. Реализация метода плотного потока: алгоритм Фарнебака
4. Реализация метода, основанного на стереоректификации и stereoBM.
5. Оценка эффективности двух методов распознавания объектов на основе анализа полученных результатов.
6. Применение одного из вышеперечисленных методов для сканирования помещения на основе пары последовательных изображений.

5.2. Методы решения задачи

На вход имеем два последовательных изображения, сделанные с небольшим сдвигом (стереоизображения). Далее для задачи распознавания необходимо применить к ним один из вышеперечисленных методов. В результате применения выбранного метода к изображениям будет получена информация о смещении каждого пикселя (векторное поле смещения) или карта глубины, которая визуализирует глубину сцены в виде карты, где каждому пикселю соответствует его расстояние до объекта.

Поэтому, имея две фотографии с некоторым шагом, мы можем использовать один из этих методов для реконструкции комнаты, сегментировав движущиеся объекты от статического фона. Рассмотрим подробнее идею каждого метода.

Метод, основанный на стереоректификации и stereoBM

В двух изображениях, полученных на вход, могут возникать проблемы, связанные со смещением линий и непараллельностью камер. Поэтому, в первую очередь, необходимо провести стереоректификацию этих двух изображений, и в следствие чего пара изображений выравнивается относительно некоторых геометрических характеристик. Чтобы облегчить процесс сопоставления пикселей на обоих изображениях, необходимо решить эти проблемы с помощью метода предварительной обработки `stereoRectify`, что в дальнейшем улучшает точность и соответствие точек на стереоизображениях Метод работает без предварительных этапов калибровки камеры и обработки изображения, что упрощает его применение.

Далее к обработанным стереоизображениям применяется метод `StereoBM`. Это алгоритм, используемый в компьютерном зрении для нахождения соответствий или сопоставления стереоизображений. Сопоставление стереоизображений заключается в поиске соответствий между точками или некоторыми характеристиками, и `StereoBM` работает путем сравнения схожести яркости или цвета пикселей на левом и правом изображениях. Изображение делится на небольшие блоки одинакового размера, для каждого блока на левом изображении ищется похожий блок на правом, сравниваются значения их пикселей и яркости цвета (мера сходства сумма абсолютных разностей или же сумма квадратов разностей), а после нахождения соответствующего блока, алгоритм вычисляет сдвиг (горизонтальное смещение) между соответствующими пикселями.

На выходе алгоритм `StereoBM` возвращает карту смещения или карту глубины для пары стереоизображений. Карта глубины – изображение, в котором каждому пикселю соответствует глубина или расстояние до объекта. Более темные пиксели соответствуют объектам, находящимся ближе, а более светлые - объектам, находящимся дальше. Карта смещения может быть использована для отображения смещений пикселей между левым и правым изображениями, и поможет для задач, связанных с реконструкцией трехмерных объектов или визуальным сопоставлением

Метод оптического потока

Оптический поток – это метод, который вычисляет поле векторов движения для пикселей в последовательности изображений и предоставляет информацию о движении. До работы алгоритма к изображениям могут быть применена некоторая предварительная обработка: например, необходимо, чтобы изображения были одинакового размера.

Оптический поток может быть рассчитан либо для всех точек изображения, либо для каких-то выборочных точек, которые выделены с помощью алгоритмов компьютерного зрения, как особые точки. Если оптический поток рассчитывается для всех точек изображения, то такой метод называется плотным потоком. Иначе, если же оптический поток рассчитывается для выборочной группы точек, то такой алгоритм называется разреженным потоком. К методам оптического потока относят такие методы, как: блочные методы, дифференциальные методы, вариационные методы и др.

Идея методов оптического потока

Рассмотрим подробнее идею методов оптического потока. Необходимо: для каждой точки `p(x, y)` на первом изображении найти такой сдвиг Δx и Δy, чтобы этой точке советовала точка на втором изображении $p(x + \Delta x, y + \Delta y)$. Или если `I(x,y)` – это первое изображение, то должно выполняться: `I(x,y)` $\approx$ `I(x + Δx, y + Δy)`

Пусть будем считать, что сдвиг небольшой. Тогда можем применить разложение в ряд Тейлора в окрестности точки `(x, y)` в данный момент времени `t`. Для примера рассмотрим линейное приближение:

$$\mathrm{I}(x + \Delta x, y + \Delta y, t + \Delta t) \approx \mathrm{I}(x, y, t) + \frac{\partial \mathrm{I}}{\partial x}\Delta x + \frac{\partial \mathrm{I}}{\partial y}\Delta y + \frac{\partial \mathrm{I}}{\partial t}\Delta t$$

Тогда для того, чтобы $\mathrm{I}(x, y) \approx \mathrm{I}(x + \Delta x, y + \Delta y)$, необходимо:

$$\frac{\partial \mathrm{I}}{\partial x}\Delta x + \frac{\partial \mathrm{I}}{\partial y}\Delta y + \frac{\partial \mathrm{I}}{\partial t}\Delta t$$

Разделим уравнение на Δt:

$$\frac{\partial \mathrm{I}}{\partial x}\frac{\Delta x}{\Delta t} + \frac{\partial \mathrm{I}}{\partial y}\frac{\Delta y}{\Delta t} + \frac{\partial \mathrm{I}}{\partial t} = 0, \text{где} \quad (1)$$

$v_x = \frac{\Delta x}{\Delta t}$ – горизонатльная компонента скорости

$v_y = \frac{\Delta y}{\Delta t}$ – вертикальная компонента скорости

$\frac{\partial \mathrm{I}}{\partial x}, \frac{\partial \mathrm{I}}{\partial y}, \frac{\partial \mathrm{I}}{\partial t}$ – производные изображения для соответсвующих направлений (изменение интенсивности пикселей)

Для полученного уравнения (1) имеем две неизвестных v_x и v_y. Таким образом, методы оптического потока отличаются способом нахождения неизвестных v_x и v_y: некоторые методы используют разложение в ряд Тейлора, учитывая члены более высокого порядка, другие методы используют аппрокисмацию неизвестных параметров и т.д.

Алгоритм Фарнебака

В дальнейшем для вычисления плотного оптического потока, будем использовать один из дифференциальных методов, а именно алгоритм Фарнебака. Алгоритм вычисляет оптический поток в виде векторного поля, где каждый вектор имеет две компоненты: горизонтальную и вертикальную скорость движения в соответствующем направлении. Важно, чтобы последовательные кадры изображения имели одинаковую интенсивность, и на обоих изображениях освещение не менялась.

Будем считать, что алгоритм отработал корректно, если функция вернула корректные значения оптического поток и массивы, в которых хранятся компоненты горизонтальной и вертикальной скорости, имеют ту же размерность, что и входные изображения. Также можно визуализировать оптический поток, чтобы убедиться в его корректности. Это поможет увидеть направления движения для каждого пикселя.

Основная идея алгоритма Фарнебака заключается в использовании дифференциальных уравнений для описания изменения яркости изображения в зависимости от движения объектов на последовательных кадрах. Дифференциальные уравнения оптического потока связывают изменение интенсивности, и эти уравнения решаются итеративно с помощью аппроксимации неизвестных параметров. Вместо аппроксимации линейной функцией, как в методах первого порядка, алгоритм Фарнебака использует полином второй степени, чтобы более точно описать изменение интенсивности пикселей в данной области.

5.3. Описание программной реализации

Программная реализация выполнена с использованием Python3 и интегрированной среды разработки PyCharm. Полная реализация методов представлена в [Приложение А - Приложение С]

Метод, основанный на стереоректификации и stereoBM

Реализация метода была выполнена в функции `Method_StereoDepthMap(img1,img2)`.

Опишем алгоритм для данного метода:

0. Функция на вход принимает два последовательных изображения.
1. Для обнаружения ключевых точек сопоставляются два изображения с использованием алгоритма `ORB (Oriented FAST and Rotated BRIEF)`, и в случае, если алгоритм не обнаруживает ключевых точек, то функция завершает свою работы без обработки кадров с выводом сообщения «`No keypoints or descriptors found`». После применения алгоритма `ORB`, мы получаем ключевые точки и их дескрипторы для каждого изображения. Дескрипторы ключевых точек - компактный способ представления характеристик окрестности каждой ключевой точки на изображении, они описывают окрестность каждой ключевой точки, и являются уникальными для каждой из них.
2. Дескрипторы ключевых точек сравниваются для двух изображений, и между ними находятся наилучшие соответствия, которые после сортируются и сохраняются в новой переменной.
3. Далее мы объединяем координаты соответствующих ключевых точек в одну матрицу, и с помощью функции `cv2.findFundamentalMat` находим фундаментальную матрицу и маску.
4. Теперь у нас есть информация о правильных соответствиях ключевых точек, и мы ректифицируем изображения методом `stereoRectify`. Результатом ректификации являются преобразованные изображения с прямыми эпиполярными линиями, а соответствующие матрицы преобразования сохраняются в переменных `left_rectification_matrix` и `right_rectification_matrix`. Затем проверяется успешность обработки изображений: что эти матрицы не являются пустыми, в противном случае алгоритм завершает работу с сообщением «`Stereo`

rectification failed». А получить сами изображения мы можем, применив к исходным изображениям матрицы преобразования.

5. Остается применить к обработанным изображениям метод StereoBM_create, который возвращает на выходе результат обработки: карту диспаратности, содержащую информацию о различиях между соответствующими точками на двух изображениях. По ней можно определить меру сдвига между пикселями, а именно: диспаратность представляет собой горизонтальное расстояние между пикселями в сцене, которое позволяет получить представление о глубине объектов в сцене. Чем больше значение диспаратности для пикселя, тем дальше находится соответствующий объект от камеры

Метод плотного потока

Реализация метода была выполнена в функции Method_OpticalFlowDepthMap (img1,img2). Внутри функции Method_OpticalFlowDepthMap с помощью метода calcOpticalFlowFarneback вычисляется оптический поток между двумя изображениями. На вход для метода calcOpticalFlowFarneback указаны следующие параметры:

1. img1,img2 — входные изображения для вычисления оптического потока
2. None – параметр, указывающий на пространственную размерность выходного оптического потока (значение: None, чтобы размерность сохранялась такая же, как у входных изображений)
3. float 0.05 — параметр, определяющий меру пространственного разрешения оптического потока
4. int 1 — параметр, определяющий меру временного разрешения оптического потока
5. int 12 — параметр, определяющий размер окна, используемого для сглаживания оптического потока
6. int 2 — параметр, определяющий количество уровней пирамиды для вычисления оптического потока
7. int 8— параметр, определяющий размер окна для поиска оптического потока на каждом уровне пирамиды

8. `float 1.2` – параметр, определяющий коэффициент увеличения размера пирамиды для вычисления оптического потока на каждом уровне
9. `int 0` – параметр, определяющий тип сопоставления пикселей

На выходе получаем векторный поток `flow`, первая компонента которого – горизонтальная скорость `u = flow[..., 1]`, вторая компонента – вертикальная скорость `v = flow[..., 0]`. Затем компоненты оптического потока по горизонтальной и вертикальной оси из декартовой системы преобразуем в полярные координаты, и получаем величину скорости движения (матрицу значений для каждой точки оптического потока) и матрицу значений направлений оптического потока для каждой точки.

5.4. Эксперименты

Теперь мы можем применить оба метода к одним и тем же двум изображением, которые имеют сдвиг, и сравнить результаты, полученные для них. Программная реализация вышеописанных методов см в п. [Приложение В].

Для сопоставления результатов работы вышеописанных методов будем использовать как смоделированные, так и реальные изображения комнат с целью провести сравнительный анализ полученных результатов. Все фотографии являются стереоизображениями – сделаны с некоторым шагом.

Стереоректификация

При использовании стереоректификации изображений, они могут быть повернуты или искажены. Искажение и поворот изображений могут происходить, если не удается правильно выровнять два отдельных изображения. Поэтому сравнивая их, нужно учитывать, что StereoBM был применен к стереоректификацицированному изображению, которое искажено, и на выходе также перспектива комнаты отлична от исходной (рисунок. 5):

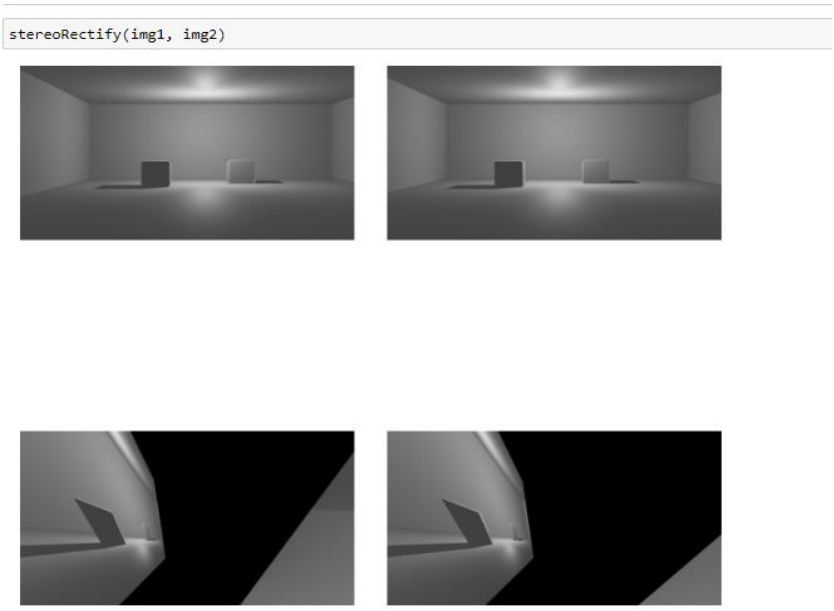


Рисунок. 5. применение стереоректификации к одной из пар изображений

Сравнение карт глубины: стереоректификация с использованием stereoBM и плотный поток Фарнебака

Обработанные фотографии будем выводить для фотографии «первый кадр». Вторая фотография сделана с шагом вправо. Применив вышеописанные методы, получим следующие результаты (рисунок. 6-7):

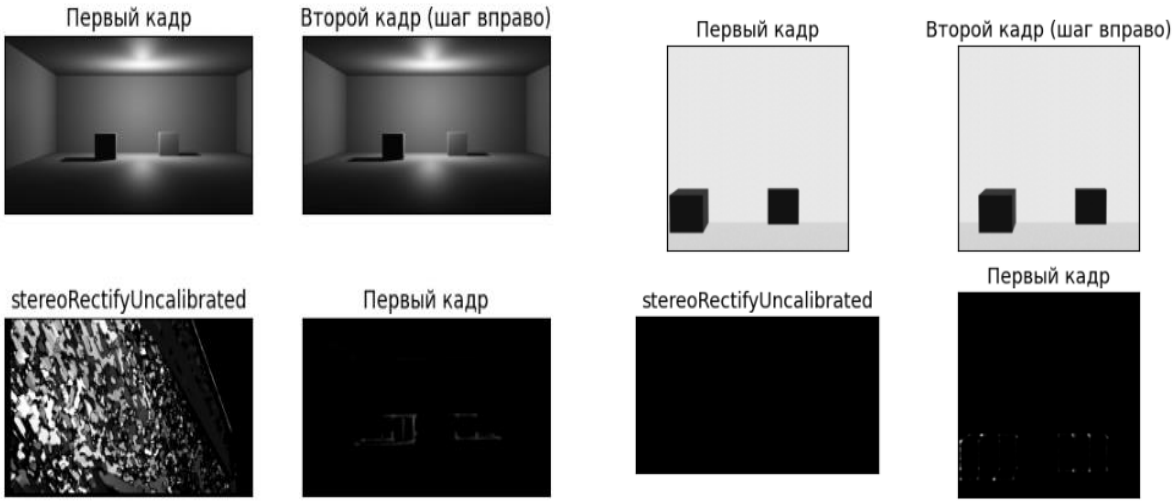


Рисунок. 6. Пример1-2: применение метода StereoBM (слева) и метода «плотного» потока (справа)

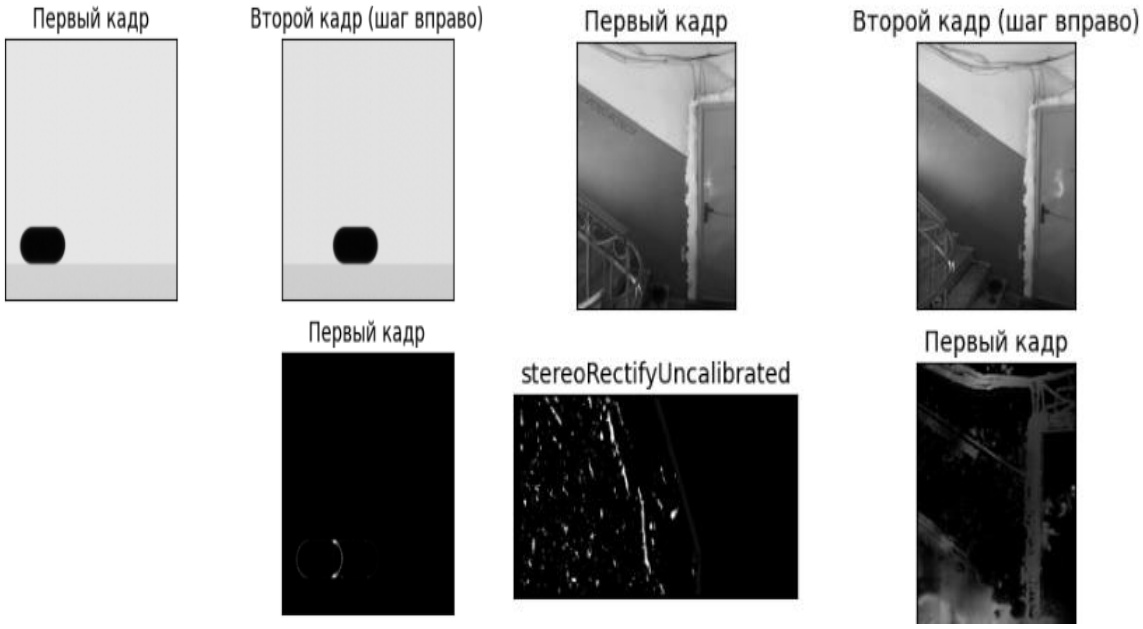


Рисунок. 7. Пример3-4: применение метода StereoBM (слева) и метода «плотного» потока (справа)

Вывод

Алгоритм Фарнебака отработал корректно и правильно выделил смещенные объекты на всех сериях стереоизображений. Однако метод StereoBM, который не использует характеристики камеры при стереоректификации, показал искаженные изображения в неправильной перспективе, что и ухудшило результаты. Поэтому требуется предварительная правильная обработка изображения перед поиском карты глубины. В данном же случае характеристики камеры неизвестны, из-за чего и получились такие результаты от применения `stereoRectifyUncalibrated + StereoBM`.

Подводя итоги: метод оптического потока показал хорошие результаты, в то время как StereoBM требует дополнительных усовершенствований, таких как правильная обработка изображения и использование характеристик камеры, чтобы добиться более точных выходных результатов для описанного метода.

5.5. Применение метода плотного потока «Фарнебака»

Пусть мы выделили объект на серии двух стереофотографий с использованием метода плотного потока. Теперь необходимо вычислить расстояние до каждого пикселя объекта, чтобы правильно его отобразить.

Если объект находится в т. A, а фотографии были сделаны в точках O и $O`$, то соединив эти три точки, получим $\Delta AOO`$. Затем от основания треугольника, проходящего через точки O и $O`$, вычислим расстояние до точки A (глубина, насколько далеко точка от камеры).

Введем обозначения:

1. $B = |OO`|$ - шаг между двумя стереоизображениями
2. w - ширина изображения ($w_1 = w_2 = w$, где w_1 и w_2 ширина двух стереоизображений)
3. h – высота картинки ($h_1 = h_2 = h$, где h_1 и h_2 высота двух стереоизображений)
4. f – фокусное расстояние: характеристика камеры, определяющая, как сильно лучи света сходятся/ расходятся после прохождения через объектив камеры и ее линзы (маленькое значение фокусного расстояния означает большой угол обзора и широкое поле зрения, а большое значение фокусного расстояния обеспечивает узкое поле зрения и большое увеличение)
5. X, Y – координаты т. A (Ox и Oy направлены по ширине и длине картинки)
6. Z – расстояние до объекта (Oz направлена вглубь картинки)

Рисунок. 8. Подобие треугольников в плоскости Oxz

Пусть примем, что расстояние между двумя фотографиями $B = 1$. Мы так можем сделать, т.к. визуализируем комнату относительно, без точных измерений расстояния. Пропорции всех предметов должны изменяться в одном и том же соотношении.

Фокусное расстояние нужно для вычисления расстояния Z (т.е. оно влияет на вытянутость фотографии вдоль оси Oz). Т.к. характеристики камеры не известны – подберем этот параметр так, чтобы масштаб комнаты казался пропорциональным. В программной реализации параметр f может принимать произвольное значение, не оказывая влияния на процесс реконструкции комнаты. В качестве примера, выберем f = 300.

Вывод формул для реконструкции 3D-координат из оптического потока

Точка A - точка в 3D-пространстве на пересечении лучей из центра камеры O и O` и имеет координаты (X, Y, Z) в системе координат OXYZ левой камеры. Тогда для системы координат правой O`X`Y`Z` камеры т. A имеет координаты (X`, Y`, Z`), где:

$$X` = X - B;$$

$$Y = Y`;$$

$$Z = Z`$$

Камера O проецирует точку A в точку $\tilde{X}$, камера O` проецирует точку A в точку $\tilde{X}$`. Тогда x и x` - координаты проекции точки $\tilde{X}$ и $\tilde{X}$` на изображение левой и правой камеры соответственно.

Разница между координатами проекции точки $\widetilde{X}$ и $\widetilde{X}$` на изображение левой и правой камеры называется диспаратностью δ:

$$\delta = x - x`$$

По подобия двух треугольников ΔOBA и $\Delta OF\tilde{X}$: $\frac{x}{f} = \frac{X}{Z} \Leftrightarrow X = \frac{Zx}{f}$

Аналогично получим и для правой камеры: $\frac{x`}{f} = \frac{X}{Z} = \frac{X`-B}{Z} \Leftrightarrow X` = \frac{Zx`}{f}$

Подставим X` = $\frac{Zx`}{f}$ = X – B = $\frac{Zx}{f}$ – B.

Выразим Z из выражения $\frac{Zx`}{f} = \frac{Zx}{f}$ – B: $Z\left(\frac{x-x`}{f}\right) = B$

Откуда получаем формулы для глубины точки A: $Z = \frac{Bf}{x-x`} = \frac{Bf}{\delta}$

Теперь можем вычислить координаты X и Y точки A в системе координат левой камеры O. Но вместо использования координат x и y в пикселях, отсчитанных от верхнего левого угла изображения, мы будем использовать $(x - \frac{w}{2})$ и $(y - \frac{h}{2})$, чтобы привести координаты пикселей к системе координат, где центр изображения — это (0,0). Тогда получим:

$$X = \frac{Z\left(x - \frac{w}{2}\right)}{f}$$

$$Y = \frac{Z\left(y - \frac{h}{2}\right)}{f}$$

Таким образом, координаты до объекта (до фиксированного пикселя x, y на изображении) можно посчитать по следующим формулам:

$$Z = \frac{B \cdot f}{\Delta\delta}, X = \frac{Z(x - \frac{w}{2})}{f}, Y = \frac{Z(y - \frac{w}{2})}{f}, \text{где:} \quad (1)$$

Из формулы (1) видно, что если Z подставить в формулу для X и Y, то они не зависят от фокусного расстояния f (характеристика камеры). Тогда, подберем это значение так, чтобы масштаб оставался визуально пропорциональным для восстановленной комнаты.

Применение метода Canny

Вместо того, чтобы обрабатывать и вычислять координаты для всех пикселей исходного изображения - применим метод «Canny» (алгоритм обнаружения границ на изображении). Метод работает следующим образом:

0. Вход: черно-белое изображение с одним каналом (grayscale: каждый пиксель принимает значение от 0 до 255)
1. Фильтр Гаусса: применяется к исходному изображению для сглаживания и удаления шума.
2. Вычисление градиента сглаженного изображения: применяется для определения изменения интенсивности в яркости для каждой точки изображения.

Сглаженное изображение: $\tilde{I} = \left(\tilde{I}^{k=1}{}_{x,y}\right) = \begin{pmatrix} \tilde{I}_{1,1} & \cdots & \tilde{I}_{1,w} \\ \vdots & \ddots & \vdots \\ \tilde{I}_{h,1} & \cdots & \tilde{I}_{h,w} \end{pmatrix}$

Ядра свертки по горизонтали и вертикали соответственно: $\begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$

Далее ядра применяется к изображению и получаем горизонтальную и вертикальную составляющую градиента (два изображения, матрицы):

$$G_x, G_y \in R^{w \times h}$$

В каждой точке (x^*, y^*) изображения теперь можем вычислить значение величины градиента и его направление (направление градиента совпадает с направлением максимального изменения яркости):

$$G_{x^*,y^*} = \sqrt{{G_x}^2|_{x^*,y^*} + {G_y}^2|_{x^*,y^*}}$$

$$\alpha_{x^*,y^*} = \arctan\left(\frac{G_y|_{x^*,y^*}}{G_x|_{x^*,y^*}}\right)$$

3. Устанавливаются два порога фильтрации: нижний T_{min} и верхний T_{max}
   3.1. Если значение градиента G_{x^*,y^*} в точке (x^*, y^*) превышает верхний порог T_{max} ($G_{x^*,y^*} > T_{max}$) или меньше нижнего порога T_{min} ($G_{x^*,y^*} < T_{min}$), то точка (x^*, y^*) удаляется.
   3.2. Если G_{x^*,y^*} в точке (x^*, y^*) между нижним и верхним порогом (T_{min}, T_{max}), то пиксель (x^*, y^*) принимается за границу (при условии, что значение величины градиента в соседних точках не больше).

Запустим алгоритм `Canny` для обнаружения границ на двух изображениях смоделированной комнаты и увидим результаты его работы (рисунок. 9):

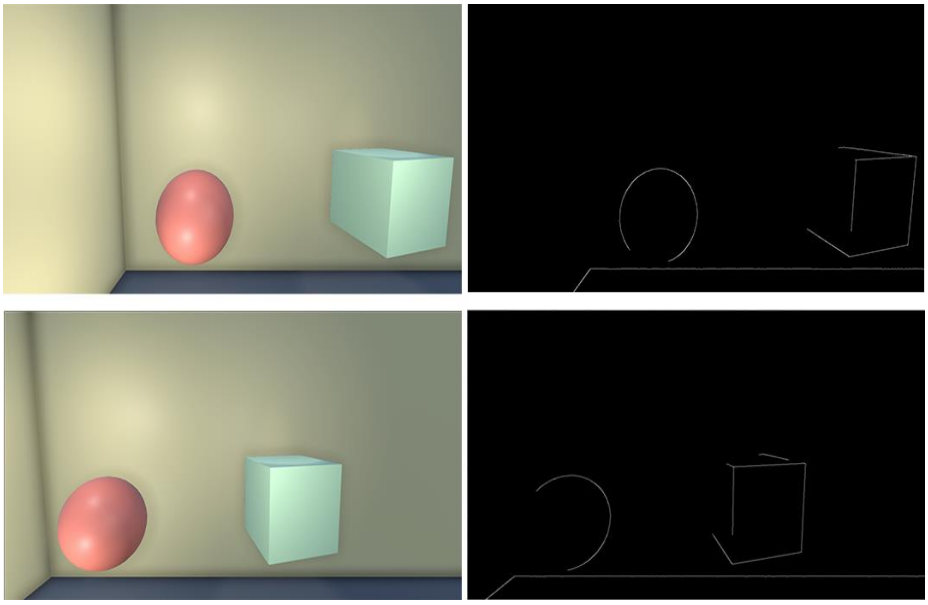


Рисунок. 9. Пример1: применение метода `Canny`.

Результат применения метода плотного потока с методом Canny

Теперь же можем применить метод плотного потока и вышеописанные формулы, чтобы получить координаты точек объектов комнаты с фотографии. В программной реализации каждая точка для фиксированного пикселя записывается в список по порядку (X, Y, Z) и каждый такой список добавляется в массив точек `points`. Все такие точки можем визуализировать – сохраним их в формат ply и увидим результат работы (рисунок. 10):

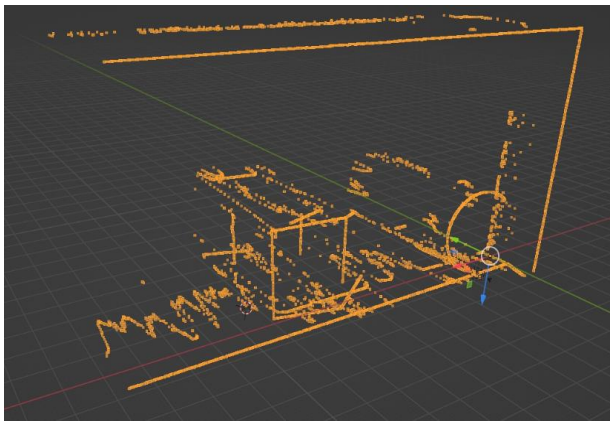


Рисунок. 10. Пример1: визуализация комнаты по точкам

Рассмотрим также результат работы программной реализации, отфильтровав шумовые точки (рисунок 11-12)

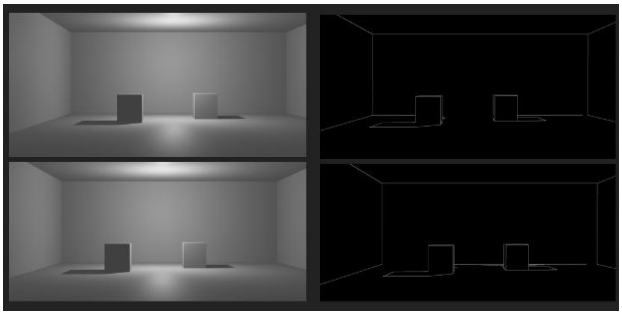


Рисунок. 11. П ример2: применение метода `Canny`

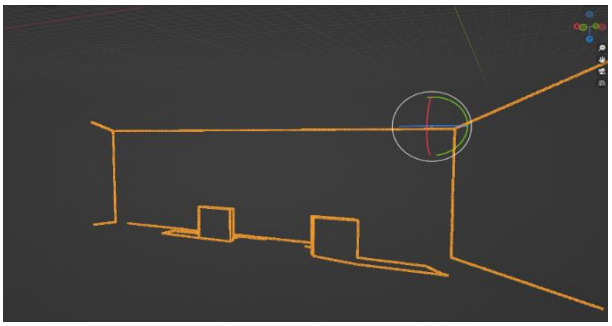


Рисунок. 12. Пример2: визуализация комнаты по точкам (без шумовых точек)

Применим вышеописанный алгоритм и для реальной фотографии, запустим программную реализацию и увидим результат (рисунок. 13):

Рисунок. 13. Пример3: результат применения метода Canny

Метод Canny отработал корректно и показал свою эффективность даже для таких сложных границ на реальной фотографии. Теперь рассчитаем расстояние до точек, сгенерируем массив точек и визуализируем в Blender (рисунок. 14):

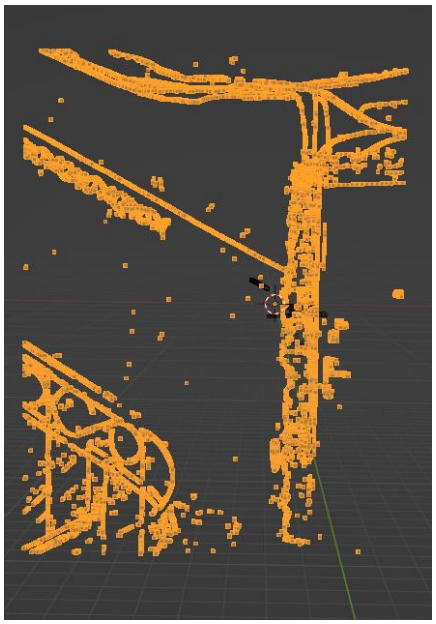


Рисунок. 14. Пример3: визуализация по точкам

Таким образом, на обоих примерах, на смоделированных и реальных стереоизображениях, получили множество точек, практически в точности, реконструирующие изображение. Однако из-за влияния света во втором примере и метод Canny выделил тень, как отдельный объект, и было вычислено расстояние до соответствующих линий, и точки этих прямых входят в массив `points`. Метод успешно отработал, и массив `points` содержит корректные данные изображения. Также в программной реализации имеются проверки на корректность ввода и этапов предобработки, и в случае возникновения ошибок, будет выведено соответствующее сообщение об ошибке, что позволяет убедиться в корректности работы метода. Полная реализация визуализации комнаты представлена в [Приложение С. Визуализация комнаты (по точкам)].

Глава 6. Кластеризация и классификация массива точек после применения метода оптического потока

Ранее было проведено сравнение метода оптического потока с методом, основанным на применении алгоритма ректификации и алгоритма поиска карты глубины по парам стереоизображений (метода «плотного» потока Гуннара Фарнебака и метода «`stereoRectifyUncalibrated + StereoBM`»). В ходе сравнения и проведения экспериментов было получено, что метод плотного потока более эффективен в задаче распознавания изображений и демонстрирует более точные результаты на тестовых данных. Поэтому в дальнейшем исследовании будем работать именно с методом плотного потока Гуннара Фарнебака.

Пусть имеем два стереоизображения, к которым был применен метод плотного потока. По формулам, описанным в п. [5.4.] $Z = \frac{B \cdot f}{\Delta x}$, $X = \frac{Z(x - \frac{w}{2})}{f}$, $Y = \frac{Z(y - \frac{w}{2})}{f}$, найдем координаты особых точек на изображении. Так, получим массив, в котором покоординатно (X, Y, Z) хранятся найденные точки.

Для дальнейшей реконструкции помещения с имеющихся фотографий – необходимо определить распознанные объекты, а именно провести кластеризацию и классификацию. Выполнив кластеризацию и разбив облако точек на подобласти различных объектов, проще провести классификацию распознанных предметов с фотографии. Для классификации объекта в каждом кластере можем использовать алгоритмы с учителем, либо без учителя.

Так, например, для кластеризации объектов на основе облака точек в трехмерном пространстве можно использовать методы машинного обучения, а именно:

1. Метод k ближайших соседей, используемый для разделения набора данных на заранее неизвестное количество кластеров, и основная идея которого заключается в том, что каждый кластер может быть описан как центроид (среднее значение всех точек данных в кластере).
2. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) – алгоритм, который основывается на плотности точек данных и позволяет выделять кластеры произвольных размеров, обнаруживая шумовые точки.

Таким образом, если для классификации мы будем используем алгоритм с учителем, то необходимо иметь обучающую выборку, состоящую изпар объект-метка, и на основе обучения предсказывать метки для отсканированных объектов с фотографии. К алгоритмам с учителем относят, например: логистическую регрессию (logistic regression), метод опорных векторов (support vector machine, SVM), деревья принятия решений и т.п. Для реконструкции помещения на основе фото данных, использование обучения с учителем может оказаться недостаточно эффективным, т.к. в комнате могут быть различные предметы и детали интерьера, для которых может быть сложно создать обучающую выборку. В комнате могут быть уникальные предметы мебели, декора или другие элементы, для которых не существует обучающих данных. В этом случае классификация с учителем, которая требует заранее определенных меток для каждого объекта, может быть неэффективной.

Поэтому вместо обучения с учителем для реконструкции помещения на основе фото данных можно использовать методы классификации без учителя, использующие алгоритмы компьютерного зрения, алгоритмы структурирования изображений и другие алгоритмы, способные автоматически анализировать и извлекать информацию из изображений без предварительно размеченных данных. Такие методы позволяют более гибко и эффективно работать с разнообразными объектами и деталями интерьера, делая процесс реконструкции более точным и автоматизированным.

Таким образом, для кластеризации объектов на основе облака точек в трехмерном пространстве выбран алгоритм DBSCAN и для классификации распознанных предметов с фотографии будет использоваться алгоритмы без учителя. Так, последовательное применение методов кластеризации и классификации на основе выбранных алгоритмов позволит эффективно обработать данные и достичь поставленной цели исследования.

6.1. Кластеризация: применение метода DBSCAN

Методы решения

Идея алгоритма заключается в обнаружении характерного скопления точек внутри каждого кластера, которая существенно превышает плотность вне кластера. Более того, в областях с шумом плотность точек ниже плотности любого из кластеров. Т.е. алгоритм учитывает не только плотность скопления точек внутри выделенного кластера, но и различия в плотностях точек между кластерами и областями с шумом. Таким образом, для каждой точки в кластере соседи в заданном радиусе должны включать не менее определенного количества точек, которое задается пороговым значением.

На вход алгоритм принимает два параметра: радиус ε −окрестности (определяет область вокруг точки, в которой ищутся для нее ищутся соседи) и m – минимальное количество соседей для формируемого кластера.

Оптимально подобранные гиперпараметры для поставленной задачи позволяют алгоритму избегать образование кластеров маленьких размеров из-за недостаточной плотности точек внутри него. Если же для поставленной задачи гиперпараметры выбраны слишком большими, то это приводит к неверному объединению нескольких различных кластеров в один. Более того, если среди облака точек есть наличие шума - алгоритм эти точки может добавить в кластер другого класса, что соответственно искажает результаты.

Пусть задана симметричная функция расстояния $\rho(x, y)$. Заданы параметры $, m = const$. Введем несколько определений для описания алгоритма и методов его работы для задачи кластеризации набора точек:

1. ε-окрестность точки x (объекта x) - область $E(x)$: $\forall y \in E(x)$ выполняется $\rho(x, y) < \varepsilon$
2. Корневой объект степени m – такая т. x (объект): $|E(x)| \geq m$ ε (т.е. это такая т. x, что ее ε-окрестность содержит не менее m объектов)
3. Точка (объект) $\tilde{x}$ непосредственно плотно-достижим из т. (объекта) x, если $\tilde{x}$ $\in$ E(x) и x является корневым объектом
4. Точка (объект) $\tilde{x}$ плотно-достижима из т. (объекта) x, если $\exists\, p_1, \dots, p_n; p_1 = x, p_n = \tilde{x}$: $\forall\, k \in 1, \dots, n$ выполняется $p_{i+1} \in \mathrm{E}(p_i)$ и p_i – корневой объект (т.е. p_{i+1}непосредственно плотно-достижима из p_i)

Алгоритм DBSCAN:

1. Выберем корневой объект x из всего множества точек
2. Пусть $x_1, \dots, x_k$ – непосредственно плотно-достижимые точки из т. X
3. $\forall\, i = \overline{1,k}$ поверяем, корневая ли точки x_i.
4. Если т. x_i корневая, то добавим всех соседей данной точки в список для обхода.

Псевдокод DBSCAN:

0. На вход поступает набор точек P, полученных после применения метода оптического потока, параметры ε и m, и функцию расстояния
1. Инициализируются массивы для шумовых точек, для посещенных точек и для кластеризованных точек
2. Для каждой точки из множества P проверяется, нет ли ее в списке посещенных:
   Если точка уже была посещена, то пропускаем ее
   Иначе помечаем точку, как посещенную (добавляем в массив посещенных точек) и находим для нее соседей
   2.1. Если количество соседей меньше m, то добавляем точку в кластер шум
   Иначе: обновляем кластер:
   2.1.1. Проверяется, принадлежит ли кластеру точка p. Если нет, то добавляем p к кластеру и точка p помечается, как принадлежащая кластеру
   2.1.2. Цикл пока для точки p есть необработанные соседи, внутри которого для каждого соседа находятся его соседи с дальнейшим определением в тот же кластер, что и у точки p, либо в кластер для шумовых точек

Таким образом, кластеры, образованные помеченными точками в процессе алгоритма, являются максимальными, т.е. их нельзя расширить добавлением другой точки. Если обошли не все точки, то можем перезапустить алгоритм из другого корневого объекта и ни один из новых кластеров не будет содержать в себе кластер с предыдущего обхода набора облака точек. Главные недостатки алгоритма DBSCAN — неспособность соединять кластеры через пропуски с меньшим расстоянием, чем ε. А также — склонность к объединению явно различных кластеров. Поэтому при увеличении размерности данных N, становится больше областей, где некорректно возникают пропуски или лишние связи между двумя областями.

Кластеризовать проще 2D сцену. Поэтому разобьем двумерную сцену на кластеры, и чтобы сформировать кластеры для 3D сцены, будем перемещать точки внутри этих кластеров вдоль одной из осей. Таким образом, вращением 2D кривых, мы получим большие по размеру кластеры, внутри которых находятся точки, соответствующие объектам в трехмерном пространстве.

Трудоемкость кластеризации

Алгоритм DBSCAN имеет линейную временную зависимость от количества точек данных N, т.е. имеет сложность работы порядка O(N). Но в худшем случае, когда данные плохо структурированы или используется наивная реализация, алгоритм имеет квадратичную зависимость времени выполнения от количества точек данных и сложность может увеличиться до $O(N^2)$.

Описание программной реализации

Программная реализация кластеризации алгоритмом DBSCAN была выполнена на языке Python Полная реализация перечисленных ниже функций представлена в [Приложение Е. Алгоритм кластеризации DBSCAN].

Реализация метода была выполнена в функции `def dbscan_naive(points_set, eps, m, distance)`, принимающая на вход:

1. `points_set` – массив множества точек, полученный после применения метода оптического потока
2. `eps` – радиус ε-окрестности
3. `m` – минимальный размер кластера
4. `distance` – функция для подсчета расстояния между двумя точками

При реализации были использованы следующие вспомогательные функции:

5. `def distance2(point1, point2)` – реализация функции расстояния для двух точек, подаваемых на вход. Расстояние вычисляется, как сумма квадратов разности координат:$(\text{point1.x} - \text{point2.x})^2 + (\text{point1.y} - \text{point2.y})^2 + (\text{point1.z} - \text{point2.z})^2$

6. `def find_neighbors (points_set, point1, eps)` – поиск всех непосредственно плотно-достижимых точек (объектов) по множеству `points_set` из точки `point1` с радиусом ε-окрестности, равным `eps`.
7. `def update_cluster(p, neighbours, clusters, clustered_points, visited_points, eps, m, points_set, NOISE, C)` – реализация метода обновления кластера, а именно: обход непосредственно плотно-достижимых точек добавление новых соседей, проверка на шумовые объекты

В случае, если алгоритм применяется для двумерной сцены, то следует функцию `def distance2(point1, point2)` определить, как функцию растояния в двумерном пространстве, а именно: $(\text{point1.x} - \text{point2.x})^2 + (\text{point1.y} - \text{point2.y})^2$.

Эксперименты

Запустим алгоритм на случайном наборе точек и проверим, как алгоритм проведет кластеризацию на тестовом примере. Набор точек сгенерируем случайно для двумерного изображения в области $[-2; 3] \times [-1; 3]$. Параметры запуска радиус ε –окретсности $\varepsilon = 0.2$, минимальное количество точек в кластере m = 4. Получим следующий результат (рисунок. 15):

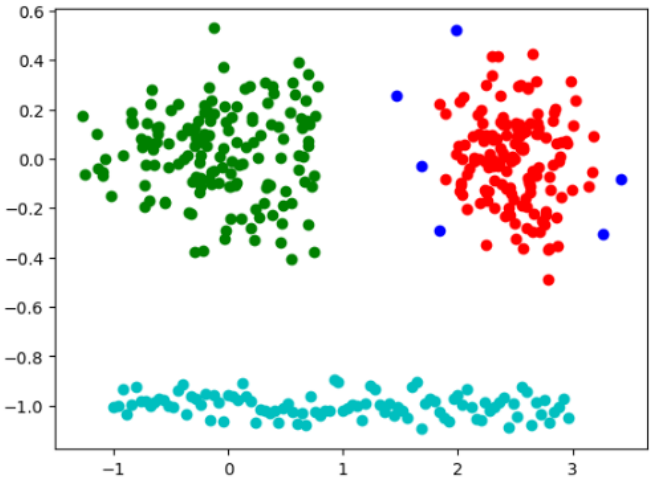


Рисунок. 15. Пример1: кластеризация алгоритмом DBSCAN

Синим цветом на изображение – это шумовые точки. Таким образом, метод DBSCAN успешно завершил свою работу и корректно обработал данные, выделив кластеры в соответствии с заданными параметрами.

Теперь запустим алгоритм на двух изображениях смоделированной комнаты и увидим результаты его работы (Рисунок. 16):

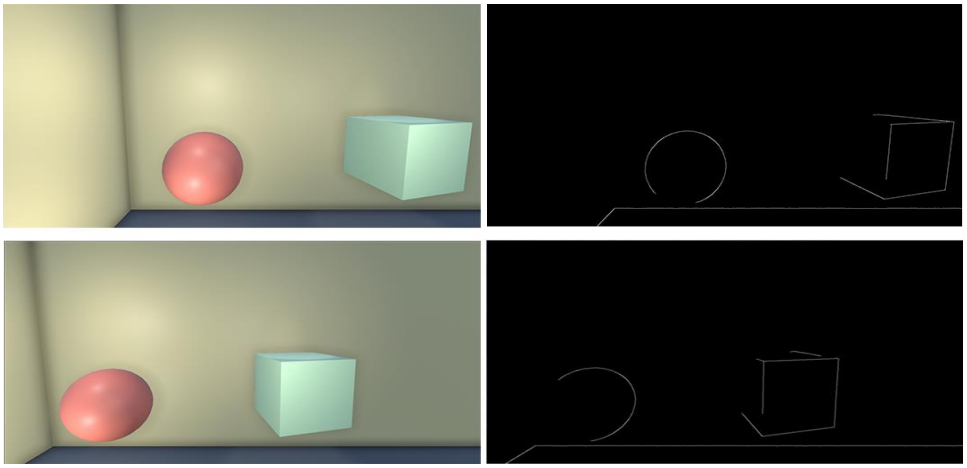


Рисунок. 16. Пример2: применение метода `Canny`.

После применения метода оптического потока имеем следующий набор точек $P = \{p_i = (x_i, y_i, z_i)\}$. Визуализируем полученное 3D-облако точек (рисунок. 17):

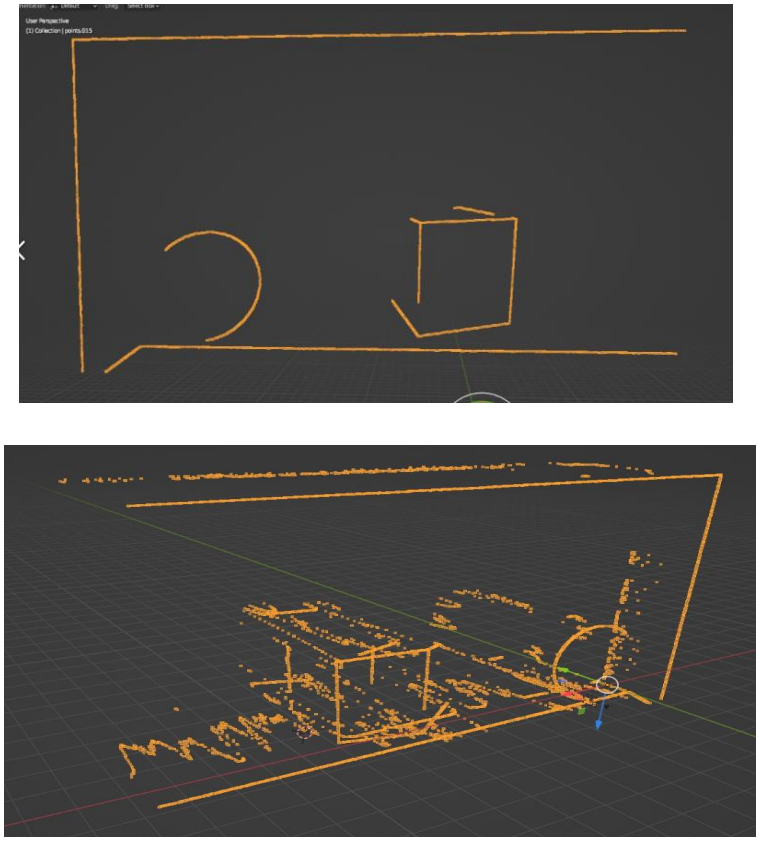


Рисунок. 17. Пример2: применение метода оптического потока (получение облака точек)

Применим алгоритм DBSCAN к массиву полученных координат, разделим облако точек на 4 кластера: пол, стены + потолок, «шар» и «куб». Таким образом, получим следующий результат (рисунок. 18-19):

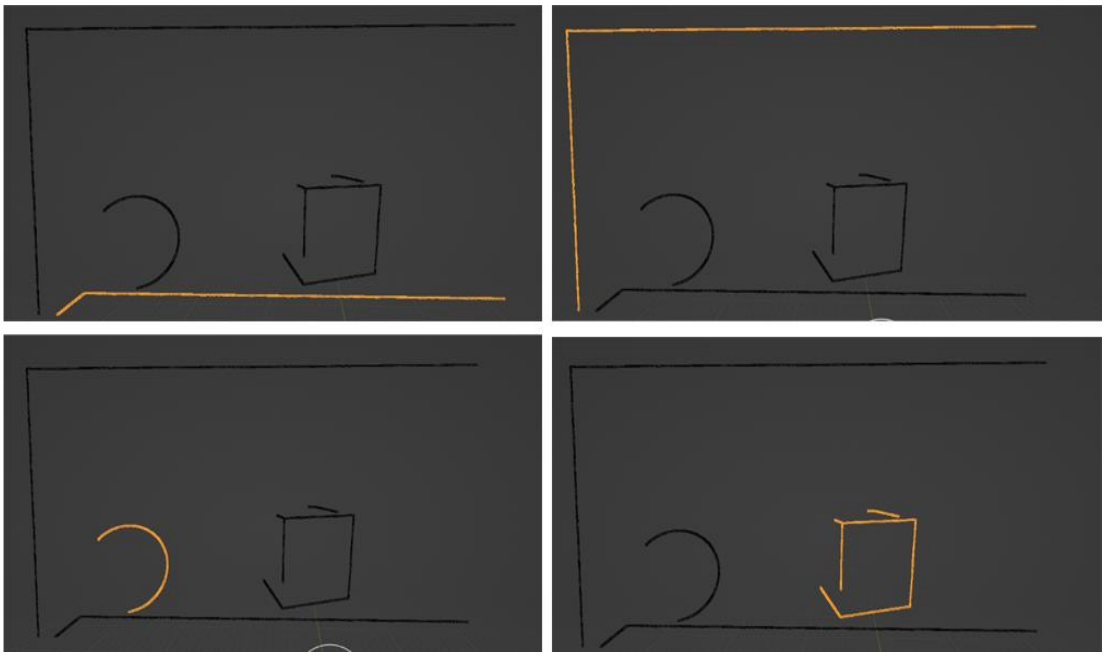


Рисунок. 18. Пример2: кластеризация облака точек методом DBSCAN

Далее рассмотрим каждый из кластеров по отдельности. Например, возьмем кластер для объекта куб (рисунок. 17 слева) и добавим к кластеру точки, которые находятся вдоль оси `Oy` (зеленая линия). Таким образом, в кластере для куба теперь хранятся все точки, которые описывает его модель в трехмерном пространстве (рисунок. 19)

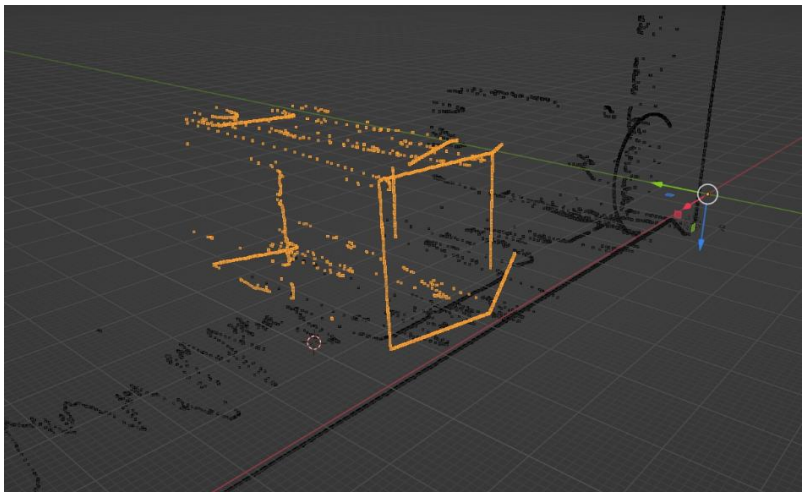


Рисунок. 19. Пример2: кластер для объекта куб - метод DBSCAN

6.2. Алгоритм классификации на основе геометрических характеристик

Алгоритм основан на анализе распределения облака точек относительно их центра масс. Сначала строится диаграмма расстояний точек до центра, а затем исследуется распределение этих расстояний в зависимости от полярного угла в полярной системе координат (угол между вектором, соединяющим центр фигуры с точкой, и положительным направлением оси Ox). Применяя свертку к диаграмме, алгоритм может выделить углы многоугольника. Если углы не обнаружены после свертки, это интерпретируется как наличие окружности в кластере. В случае обнаружения углов выполняется поиск всех углов многоугольника, и, если их количество превышает n, многоугольник не распознается алгоритмом, либо принимается за окружность.

Данный метод позволяет идентифицировать различные фигуры в двумерном облаке точек, включая окружности, треугольники, квадраты и многоугольники. Таким образом, данный метод также позволяет распознать различные геометрические формы как в двумерном, так и в трехмерном пространствах, основываясь на анализе их структуры и распределения точек. Поэтому таким способом можем классифицировать простейшие трехмерные объекты, находящиеся в определенном кластере.

Методы решения

Углы многоугольника располагаются на большем расстоянии от его центра, чем ближайшие к ним точки. Это свойство может быть использовано для определения углов многоугольника путем анализа расстояний от каждой точки до центра. Построение графика расстояний точек от центра позволяет выделить углы, поскольку они будут демонстрировать большее расстояние по сравнению с точками, находящимися на ребрах многоугольника.

Для вычисления центра многоугольника применяется среднее значение координат:

$$\bar{x} = \frac{x_1 + \cdots + x_n}{n}, \bar{y} = \frac{y_1 + \cdots + y_n}{n}$$

Затем расстояние от центра до каждой точки вычисляется по формуле:

$$\text{distance}_\text{i} = \sqrt{(x_i - \bar{x})^2 + (y_i - \bar{y})^2}$$

В дальнейшем будем считать без квадратного корня, т.к. дальше используем нормализацию подсчитанных расстояний. Далее в пункте «Эксперименты: применение алгоритма» будем использовать такое значение уже без квадратного корня. Проанализируем эти расстояния для выявления углов многоугольника: построим график расстояний distance_i в зависимости от номера точки i. Тогда можем выделить количество точек, которые подозрительны на угловые. Это более наглядно видно на графике расстояний, где значения локальных экстремумов по оси расстояния соответствуют точкам, которые могут быть угловыми.

Для улучшения визуализации графика применяется нормализация расстояний, что позволяет привести значения в диапазон от 0 до 1. Такой подход улучшает интерпретацию данных, упрощает сравнение различных графиков и делает визуализацию более понятной.

Перейдем от графика зависимости <расстояния до центра от индекса точки> `distance(i)` к графику зависимости <расстояния до центра от полярного угла относительно центра фигуры>. Эти углы будем вычислять с использованием функции `arctan2`, которая возвращает угол между положительным направлением оси `Ox` и вектором, соединяющим центр с точкой `i`. Будем по-прежнему считать, что центр многоугольника (геометрической фигуры) $\bar{x} = \frac{x_1+\cdots+x_n}{n}, \bar{y} = \frac{y_1+\cdots+y_n}{n}$. Тогда вычислим для каждой точки (`point.x`, `point.y`) искомый угол по следующей формуле:

`distance.angle = np.arctan2(delta_x, delta_y)`, где:

`delta_x = point.x -` $\bar{x}$

`delta_y = point.y -` $\bar{y}$

Функция `np.arctan2(delta_x, delta_y)` возвращает угол в радианах, лежащий в диапазоне от $-\pi$ до π, где 0 — это угол, соответствующий положительному направлению оси `Ox`, π — угол, направленный вдоль оси `Ox`, в противоположную сторону и углы между $-\pi$ и π — это все возможные направления для точек на плоскости.

Для выделения локальных экстремумов, соответствующих угловым точкам, в алгоритме используется свертка с ядром:

$$y = \sqrt{((1 - x^2) + 1\,)} - 1$$

Применяя операцию свертки к графику расстояний от углов к центру многоугольника, мы получаем новый график, на котором значения, соответствующие экстремумам, четко выделены. После свертки значения, превышающие 0 по оси ординат, соответствуют экстремальным точкам, подозрительным на угловые. Если найдено слишком много углов, то будем считать фигуру окружностью или, наоборот, добавим фильтрацию окружностей так, что если после свертки нет ни одного значения больше 1 – то фигура является окружностью.

Данный критерий не применим для обнаружения сильно вытянутых в одном направлении эллипсов, так как такие эллипсы могут иметь острые «пики», напоминающие углы многоугольника. Эллипсы и окружности можно различить по количеству обнаруженных углов: если алгоритм выявляет два или более угла, фигура, вероятнее всего, является эллипсом. Тем не менее, для задач реконструкции помещений различение эллипсов и окружностей не столь критично, поскольку более важным является правильное расположение отсканированных объектов и их приблизительное отображение. Более того, в процессе обработки облака точек применяется нормализация, позволяющая обеспечить, чтобы расстояния от центра до границы объекта находились в заданном диапазоне значений. В таких условиях окружность становится неразличимой от эллипса после нормализации. Данная нормализация упрощает классификацию объектов, так как она позволяет сравнивать их на основе формы и расположения, а не абсолютных значений координат.

Предобработка облака точек

Для классификации объекта мы используем метод, основанный на вычислении геометрических характеристик: количества углов каждой грани, расстояния до центра и т.п. Облако точек может не быть идеально замкнутым: могут присутствовать "лишние" линии или шум в данных, что делает задачу классификации объекта сложнее. Например, рассматривая одну из граней отсканированного куба, из-за ракурса, с которого была сделана фотография, во входных данных могут быть точки, образующие ребра, относящие к другой грани. Тогда таким алгоритмом, описанным выше, на графике зависимости расстояния до $\bar{x}, \bar{y}$ от точки с индексом i (индекса точки) невозможно точно определить, что рассматриваемая грань куба имеет 4 угла, т.к. лишние ребра также могут тоже образовывать угол.

Для устранения шумов удалим случайные точки из облака, чтобы снизить влияние некорректно распознанных граней и облегчить выделение основных граней объекта. Для

каждой предполагаемой грани объекта вычислим центр, как среднее координат. Далее отфильтруем точки, используя критерий расстояния от центра: точки, у которых расстояние до центра грани отличается от среднего на величину больше, чем допустимый порог (eps), считаются выбросами и удаляются. Это поможет удалить шумовые данные и улучшить форму объекта.

После очистки данных оставшиеся точки будем использовать для определения формы объекта на основе описанного алгоритма на основе геометрических характеристик. Куб, например, будет иметь грани с четырьмя углами, а сфера – гладкие грани без углов.

Описание программной реализации

Программная реализация описанного выше алгоритма классификации была выполнена на языке python. Полная реализация перечисленных ниже функций представлена в приложении: [Приложение F. Алгоритм классификации на основе геометрических характеристик]

При реализации были использованы следующие вспомогательные классы:

1. `class Point` – класс, предназначенный для представления точки в двумерном пространстве, имеющей координаты
2. `class Distance` – класс для представления характеристики точек: расстояния до центра, угол, индекс и позиция
3. `class Kernel` – класс, представляющий из себя ядро свертки, которое будет использоваться в процессах обработки данных.

   Атрибуты класса:

   - `values` - значения ядра, которые будут использоваться для свертки с данными, определяющие как сильно каждое значение данных будет учитываться при вычислении результирующего значения.
   - `Center` - индекс центрального значения в ядре, что позволяет смещать данные во время свертки

При реализации были использованы следующие вспомогательные функции:

1. `create_kernel(size)` – функция создает ядро свертки заданного размера
2. `apply_kernel(data, kernel)` – функция применяет ядро свертки к данным, вычисляя новый набор значений, которые зависят от значений входных данных и ядра. Функция принимает выходные данные `data`, к которым применяется свертка и объект `kernel` содержащий значения ядра и центр.
3. `peak_detect(data_size)` – функция для обнаружения локальных экстремумов (пиковых значений на графике индекс точки/значение свертки), принимающая на вход размер входных данных `data_size`

Для тестирования были использованы функции, генерирующие множество точек, представляющих из себя окружность, квадрат и пятиугольник.

1. `class Point` – функция для генерации точек, описывающих окружность
2. `generate_square_points` – функция для генерации точек, описывающих стороны квадрат
3. `generate_pentagon_points` – функция для генерации точек, описывающих стороны пятиугольника

Эксперименты: применение алгоритма. Проверка корректности

Для начала рассмотрим, как алгоритм работает на сгенерированной окружности, квадрате и пятиугольнике.

А. Квадрат:

Определим центр сгенерированного квадрата (рисунок. 20):

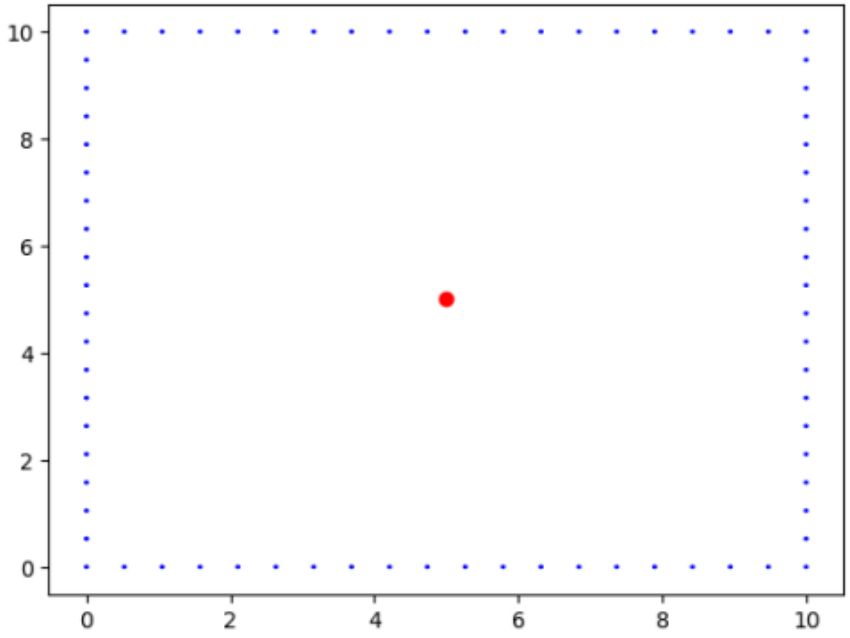


Рисунок. 20. Сгенерированный квадрат и его центр $\bar{x}, \bar{y}$

Центр находится в точке (5.0, 5.0). Квадрат описывается областью [0;10] × [0; 10]:

```
[387]: center_x, center_y

[387]: (5.0, 5.0)
```

Рисунок. 21. Центр квадрата, как среднее всех координат из облака точек для данного кластера

Построим следующие графики (рисунок. 22):

1. График зависимости расстояния до центра от индекса точки
2. График зависимости расстояния до центра от угла
3. График зависимости угла от значения свертки, примененной к расстоянию

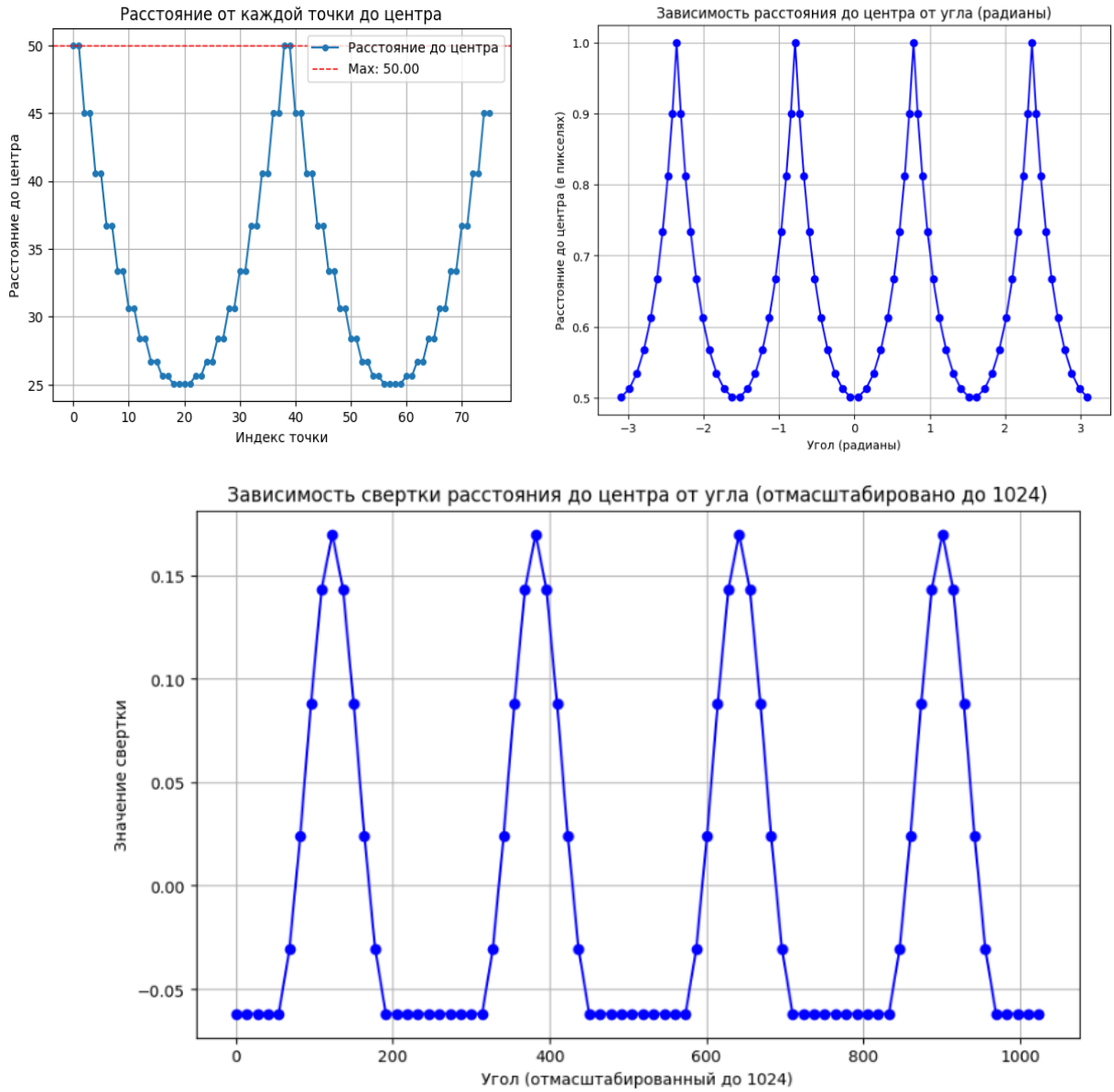


Рисунок. 22. График 1-3 для куба

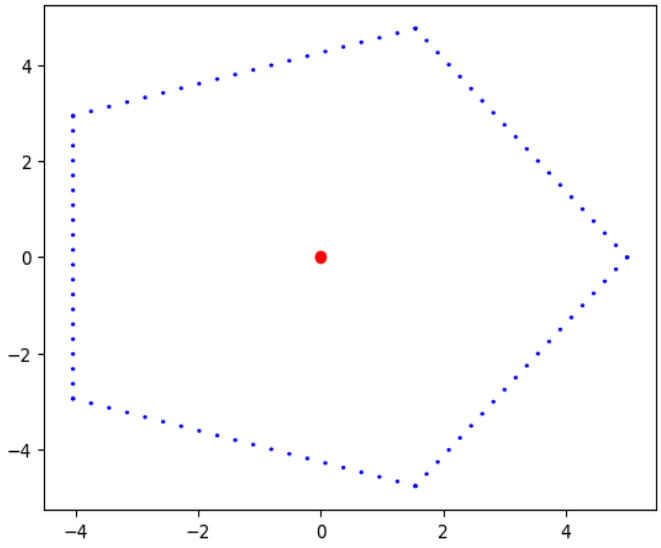
На графике 2 более наглядно демонстрируется наличие 4 углов у квадрата, чем на графике 1, поскольку индексы, соответствующие максимальному расстоянию до центра расположены близко в массиве облака точек. На практике, когда размер массива облака точек значительно превышает показанный в примере, на первом графике становится еще сложнее выявить точки, соответствующие углам. В массиве облака точек также могут встречаться точки, возникшие из-за шумов, вызванных процессом сканирования, или из-за ракурса съемки фотографии, с которой производилось сканирование.

Применив свертку (график 3), мы можем заметить, что угловым точкам соответствуют экстремумы, и все они имеют положительное значение. Эти точки можно определить с помощью алгоритма дихотомии: сначала находим точку, в которой значение свертки переходит из отрицательного в положительное, затем ищем максимум и повторяем процесс, пока не обработаем все точки.

Таким образом, алгоритм работает корректно для тестового примера квадрата. Следует отметить, что поскольку квадрат был сгенерирован для тестового примера, в массиве облака точек присутствуют четыре угловые точки. Ожидалось, что будет четыре точки с минимальным расстоянием до центра, однако, как видно из графика-1, их восемь. Это связано с тем, что в массиве отсутствуют точки, которые находятся на перпендикуляре к сторонам квадрата. На практике при сканировании помещения в облаке точек также могут отсутствовать и угловые точки, количество которых мы считаем для классификации, в следствие чего можем получить не 4 точки с максимальным расстоянием до центра квадрата, а больше. В дальнейшем мы подробнее рассмотрим эту проблему.

B. Пятиугольник:

Определим центр сгенерированного пятиугольника (рисунок. 23):




```
[126]: center_x, center_y

[126]: (np.float64(-1.3534147347811433e-16), np.float64(-4.668223479733396e-16))
```

Рисунок. 23. Центр пятиугольника

Построим аналогичные графики, как и для квадрата (рисунок. 24):

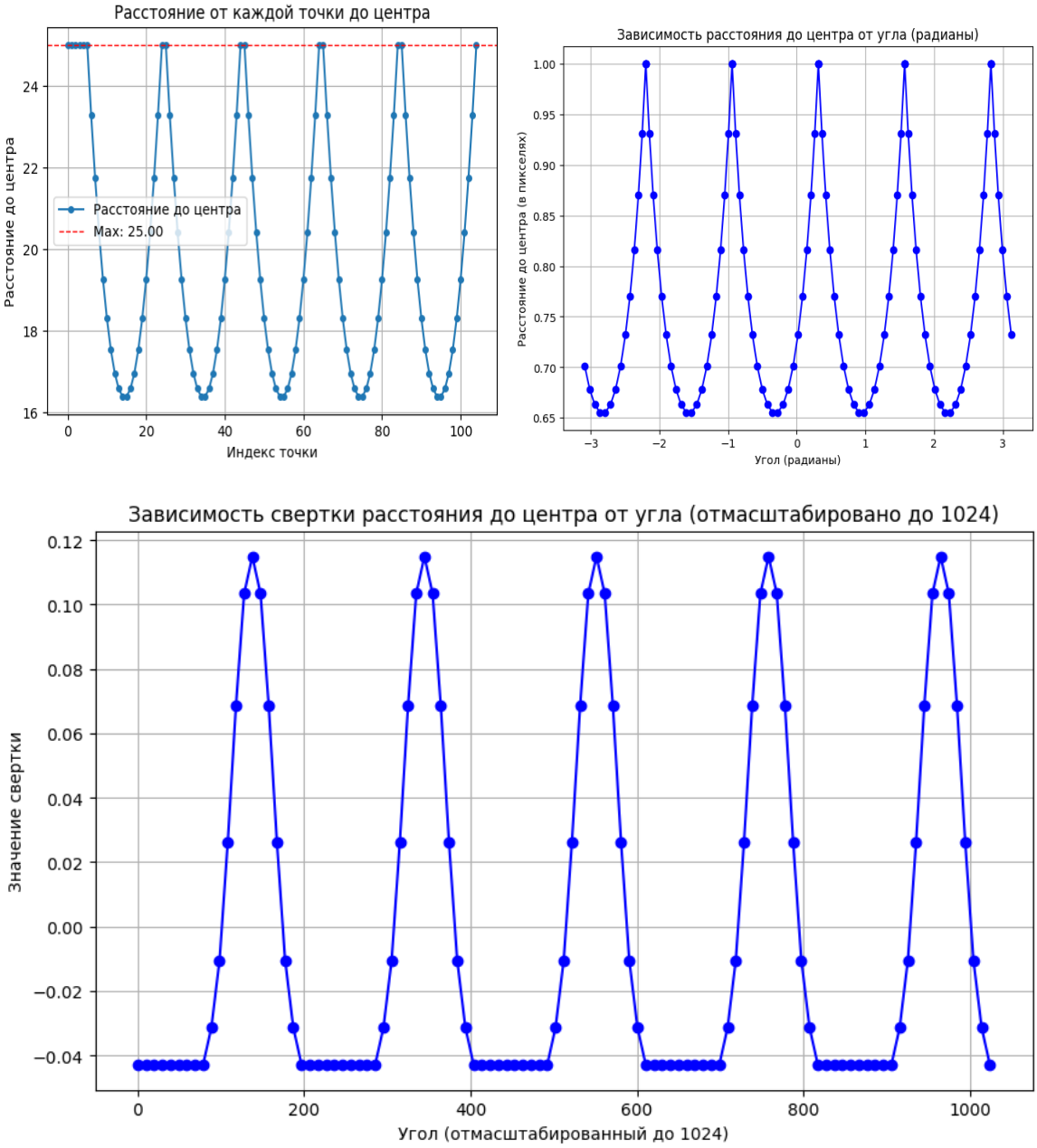
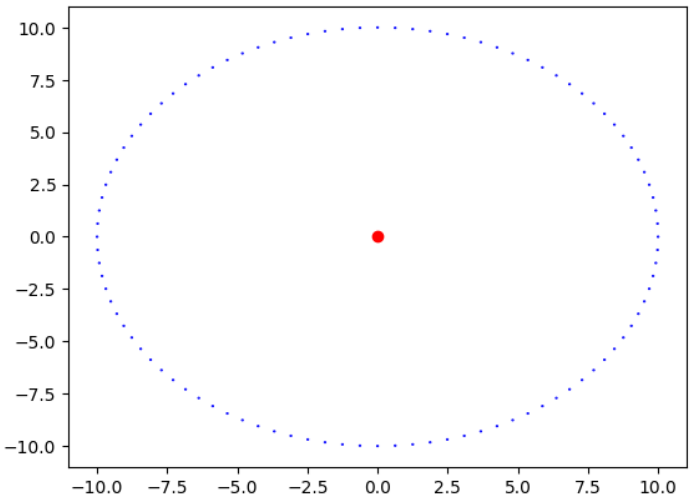


Рисунок. 24. График 1-3 для пятиугольника

Таким образом, алгоритм работает корректно и для тестового примера пятиугольника. Аналогично имеем, что на графике-2 по сравнению с графиком-1 проще обнаружить угловые точки. Применение свертки (график-3) позволяет четко идентифицировать все 5 угловых точек.

C. Окружность

Определим центр сгенерированной окружности (рисунок. 25):

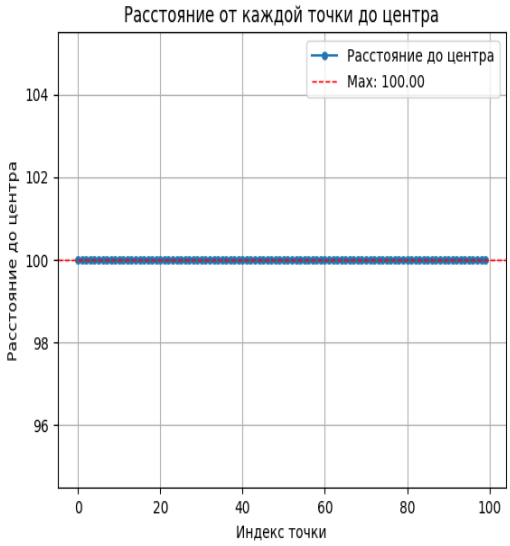



```
[155]: center_x, center_y

[155]: (np.float64(-5.329070518200751e-17), np.float64(-5.362377208939506e-16))
```

Рисунок. 25. Центр окружности

Построим аналогичные графики (рисунок. 26):



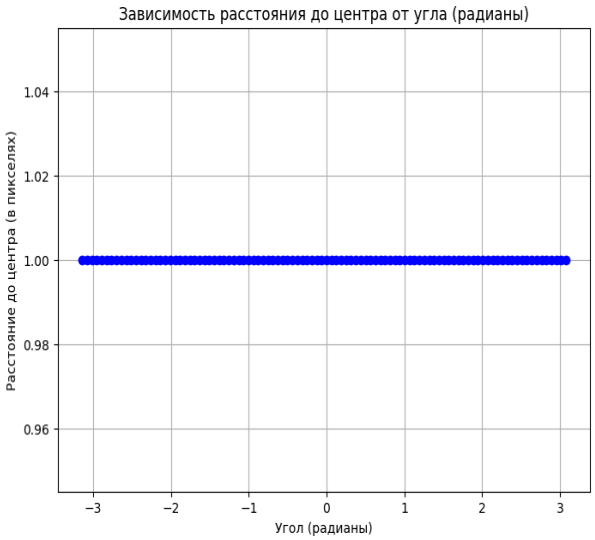
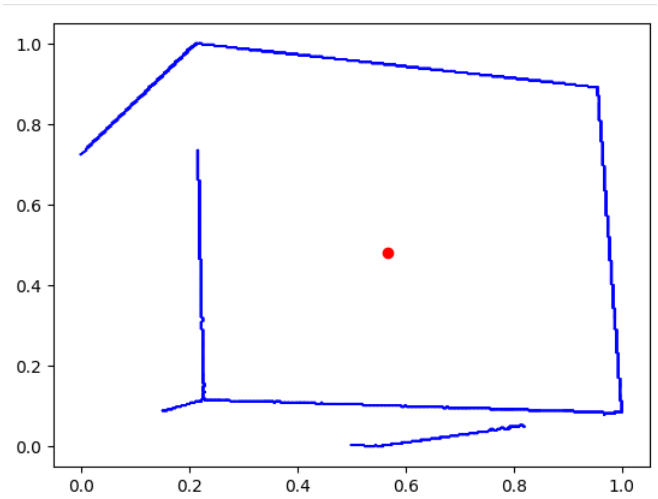


Рисунок. 26. График 1-3 для окружности

Таким образом, алгоритм работает корректно и для тестового примера окружности. Можем понять по графику-1 и графику-2, что это окружность, т.к. мы имеем график прямой с постоянным значением: т.е. в случае графика-1 расстояние для каждой точки до центра постоянно. Также можем и понять по грфику-3: большое количество локальных максимум, и мы не рассматриваем многоугольники. Вместо этого, такие фигуры будем считать окружностями.

Эксперименты: применение алгоритма. Результаты

А. Куб без обработки:
Определим центр отсканированной грани куба (Рисунок. 27):




```
[83]: center_x, center_y

[83]: (np.float64(0.5680654484430429), np.float64(0.48020847152468443))
```

Рисунок. 27. Центр грани куба из тестового примера (проекция множества точек этой грани на одну из осей)

Центр квадрата, нормализованного в диапазон [0;1] по обеим осям, оказался примерно в точке (0.5, 0.5). Построим для него график 1-3 и посчитаем количество углов (рисунок 28-29):

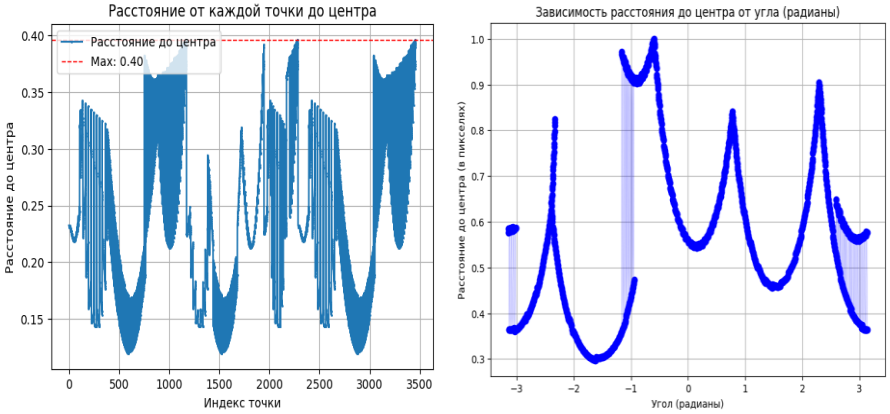


Рисунок. 28. График 1-2 для квадрата из примера

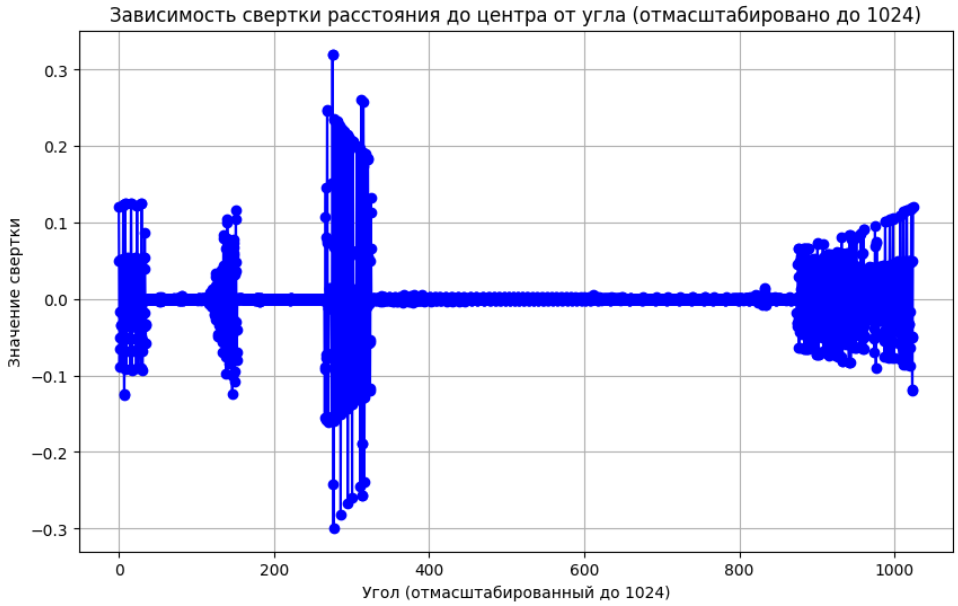


Рисунок. 29. График 3 для квадрата из примера

Таким образом, описанный выше алгоритм не смог корректно распознать форму. На графиках 1 и 2 трудно определить, сколько точек соответствуют максимальному расстоянию. На графике 3 после применения свертки оказалось, что по алгоритму количество точек, соответствующих угловым вершинам, слишком велико, что приводит к ошибочному распознаванию квадрата как окружности. Поэтому внесем некоторые изменения в алгоритм: удалим случайные точки, обработаем шумовые точки, удалим точки близкие на определённом расстоянии eps.

Если подсчитать количество точек, соответствующих максимальному расстоянию, обнаружится, что таких точек три, и их координаты практически совпадают. Это указывает на наличие в массиве точек облака очень близких друг к другу значений, где одна и та же угловая точка распознается несколько раз. Важно отметить, что эта точка действительно является угловой — верхней левой вершиной квадрата (рисунок 30):

```
count = 0
for i in range(len(distances_to_center)):
    if abs(distances_to_center[i] - max(distances_to_center)) < 0.001:
        count += 1
        print(source_points[i].x, source_points[i].y)
```

```
0.21290322580645152 1.0
0.21290322580645152 1.0
0.21290322580645152 1.0
```

Рисунок. 30. Распознанные угловые точки

В. **Куб с обработкой шумовых точек**

Сначала удалим случайные точки из куба, так как в нашем массиве облака точек их и так достаточно, и это не повлияет на общий результат. Затем удалим точки, у которых есть соседи на расстоянии eps, т.к. можем считать такие точки одинаковыми с ошибкой в одну тысячную. И затем удалим такие точки, у которых расстояние до центра отличается на eps2 от среднего расстояния до центра.

Удалим 20% точек, после чего объединим точки близкие на расстояние eps $\approx 10^{-12}$. Таким образом, после удаления 20% точек получим, что наш размер массива уменьшился с 3460 до 2768 и после объединения близких точек – размер массива составил 156. После описанных преобразовании наш квадрат выглядит следующим образом (рисунок. 31):

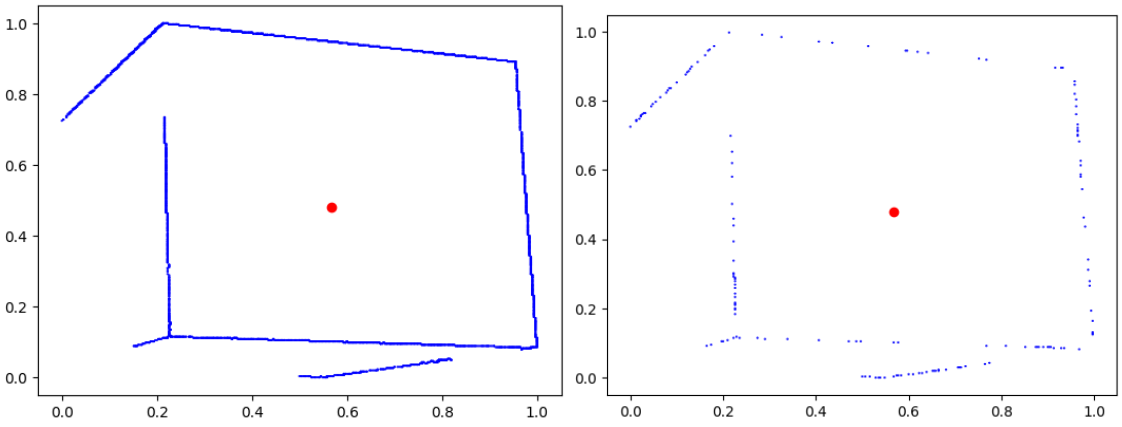


Рисунок. 31. Квадрат после описанных преобразований

Проведём эксперименты с процентом удаленных точек и со значением eps (Таблица 1):

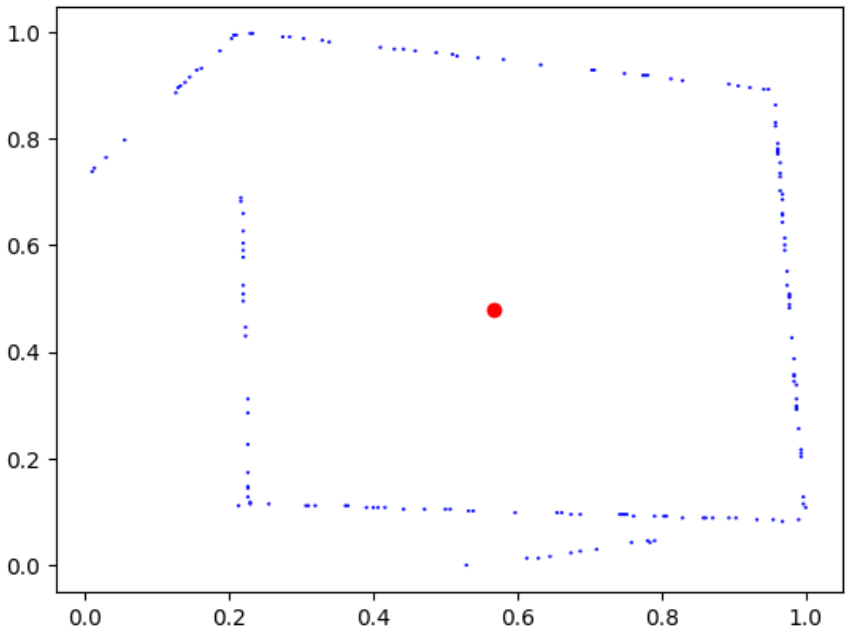
Таблица 1:

Исходное количество точек	3460	3460	3460	3460	3460	3460	3460	3460	3460	3460	3460	3460	3460
Удаление p% точек	20	20	20	50	50	50	80	80	80	95	95	95	95
Количество точек	2768	2768	2768	1730	1730	1730	692	692	692	173	173	173	173
eps	0,01	0,001	1 e-6	0,01	0,001	1 e-6	0,01	0,001	1 e-12	0,01	0,001	1 e-6	1 e-15
Количество точек	0	163	152	0	457	457	11	442	446	68	158	157	152

В результате объединения близких соседей на расстоянии eps количество точек уменьшилось до примерно 160. При использовании малого значения eps (около 0.01) в кластере точки практически полностью отсутствовали. Наилучший результат, при котором удалилось большинство точек, относящихся к другим граням куба, получился при удалении

95% точек и eps $\geq 10^{-6}$. Так как после выполнения вышеописанных операций количество точек остается примерно 160, удаление такого числа точек не оказывает значительного влияния на конечный результат. Но, с другой стороны, удалось избавиться от большего количества точек, лежащих на ребрах не рассматриваемой грани куба.

Рассмотрим алгоритм при удалении 95% точек из кластера и объединим близких соседей, на расстоянии меньшее, чем на eps = 10^{-15}. Определим центр отсканированной грани куба (рисунок 32):




```
[571]: # Расчет центра фигуры
center_x = sum([point.x for point in source_points]) / len(source_points)
center_y = sum([point.y for point in source_points]) / len(source_points)
center_x, center_y
```

```
[571]: (np.float64(0.6215788419154477), np.float64(0.4785867019651765))
```

Рисунок. 32. Центр квадрата после обработки шумовых точек

Центр исследуемой грани куба немного изменился после обработки шумовых точек, но по-прежнему остается около (0.5, 0.5). Как видно на рисунке, количество точек на рёбрах, относящихся к другим граням, уменьшилось, и теперь большую часть составляют именно рассматриваемые точки.

Построим графики 1-3 для полученного массива точек (рисунок 33):

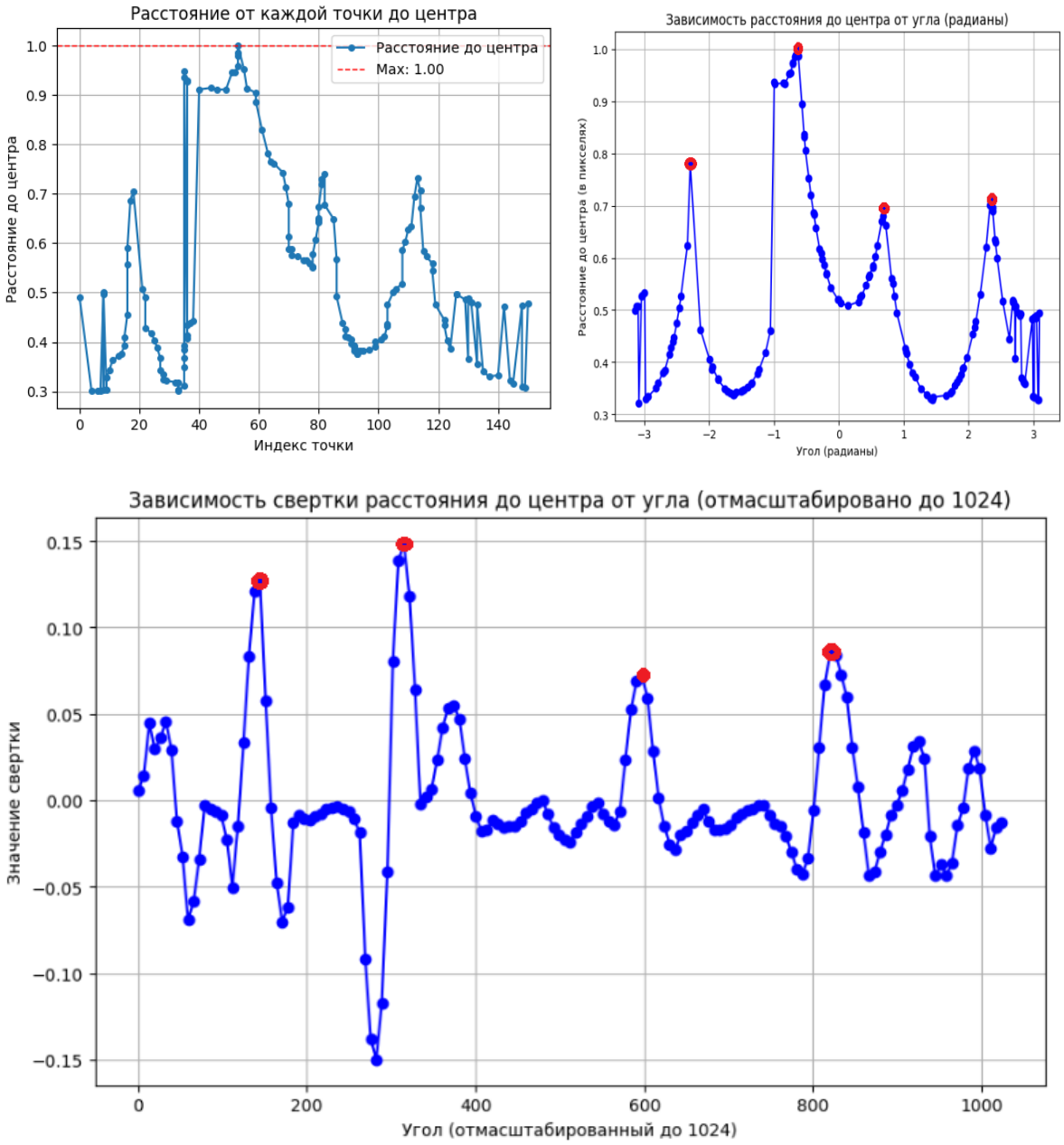
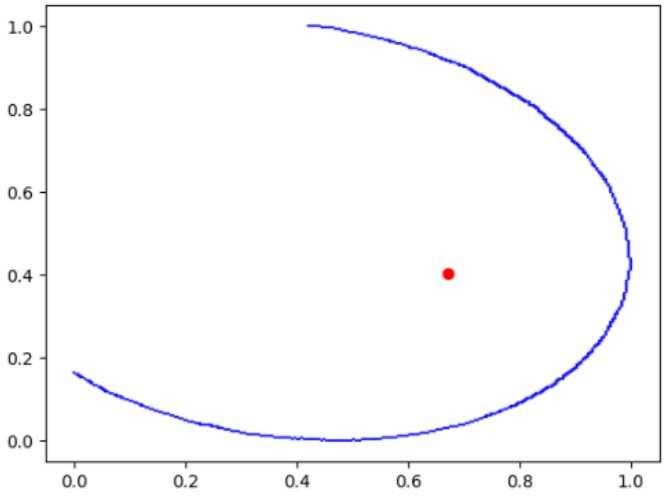


Рисунок. 33. График 1-3 для грани куба после обработки шумовых точек

Таким образом, алгоритм работает корректно для обработанного набора точек примера классификации куба. Алгоритм смог выделить 4 точки, подозрительных на угловые, по количеству которых можем сделать вывод, что рассматриваемая грань является квадратом.

C. Окружность без обработки

Построим график окружности, описываемой массивом точек, полученных после сканирования комнаты с двух изображений, и найдем ее центр (рисунок 34):




```
center_x, center_y
```

```
(np.float64(0.6726211223422794), np.float64(0.40203983620646083))
```

Рисунок. 34. Центр окружности, являющейся срезом сферы (проекция множества точек этой окружности на одну из осей)

Получаем, что центр находится в точке (0.67; 0.4). Т.к. массив точек нормирован по координатам к области [0; 1] × [0; 1], то ожидалось, что центр будет в точке (0.5; 0.5). Построим график 1-3 и попробуем классифицировать отсканированный объект (рисунок 35-36):

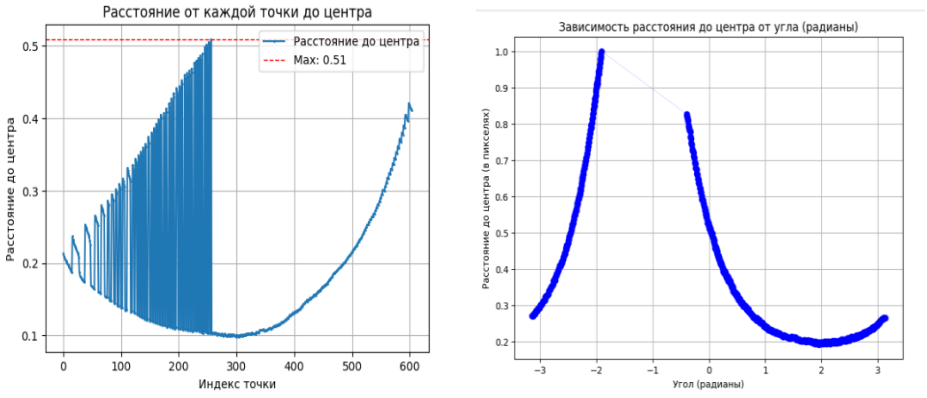


Рисунок. 35. График 1-2 для окружности

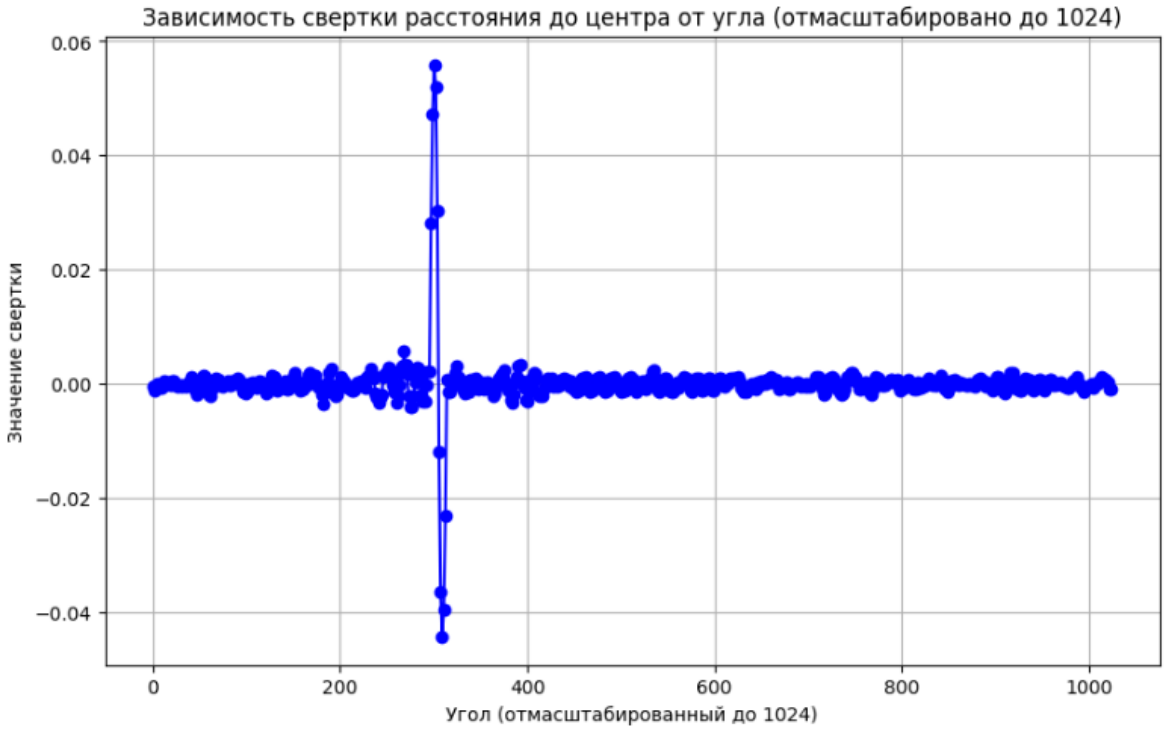


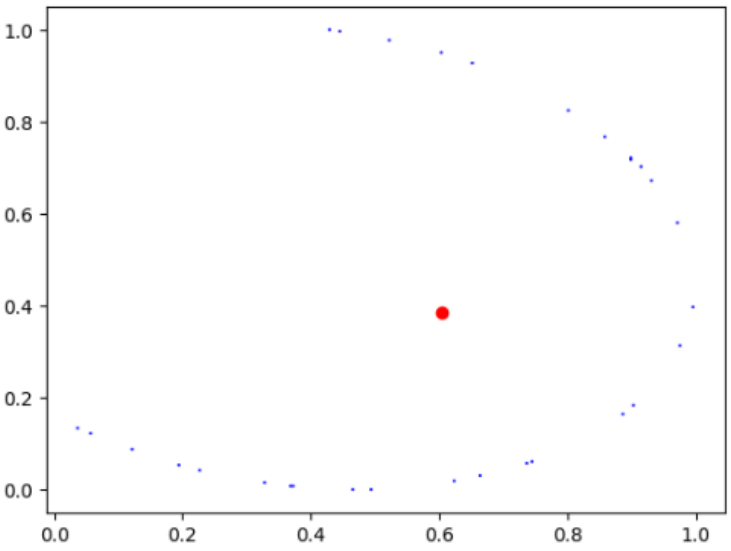
Рисунок. 36. График 3 для окружности

Поскольку окружность не имеет углов, алгоритм не смог выявить дополнительные точки с максимальным расстоянием до центра среди остальных. Такой объект следует классифицировать как окружность, так как массив значений свертки содержит единственный локальный максимум, который больше нуля.

Таким образом, описанным алгоритмом удалось распознать окружность из практического примера без обработки облака точек на шумовые точки. Далее рассмотрим, какой результат классификации будет, если применить к окружности ту же обработку, которая была использована для квадрата выше.

D. Окружность после обработки:

Для классификации окружности применим описанные методы обработки облака точек с теми же параметрами. Удалим 95% точек и после объединим близких соседей, находящихся на расстоянии меньшее, чем eps = 10^{-15}. После этих преобразований окружность выглядит следующим образом (рисунок 37):




```
# Расчет центра фигуры
center_x = sum([point.x for point in source_points]) / len(source_points)
center_y = sum([point.y for point in source_points]) / len(source_points)
center_x, center_y
```

```
(np.float64(0.6040485829959512), np.float64(0.384351145038168))
```

Рисунок. 37. Центр окружности после обработки шумовых точек

Как видно на рисунке, количество точек уменьшилось, что упрощает классификацию объекта. Мы удалили дублирующиеся и шумовые точки на основе их расстояний до центра. Теперь построим график 1-3 для данной окружности после применения всех описанных выше преобразовании (рисунок 38-39):

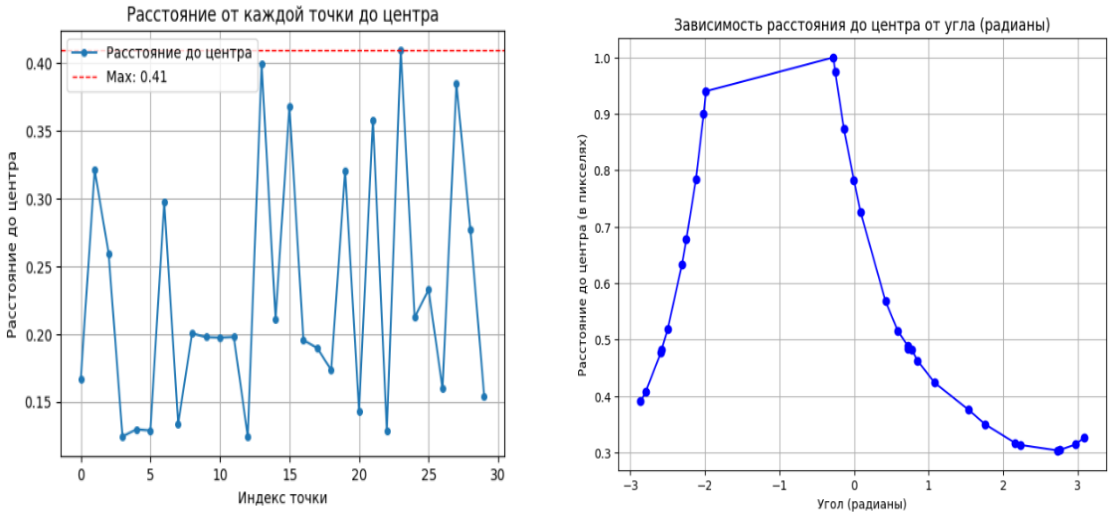


Рисунок. 38. График 1-2 для окружности после обработки множества точек

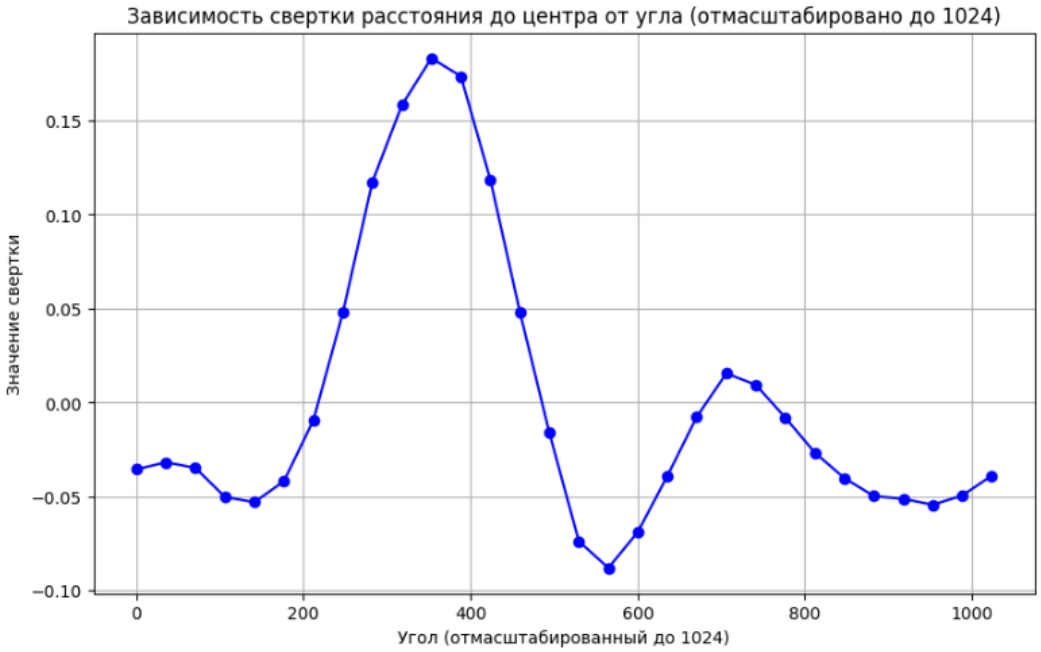


Рисунок. 39. График 3 для окружности после обработки множества точек

По графику видим, что точка подозрительная на угловую единственная. Таким образом, наша фигура является окружностью. Алгоритм работает корректно и правильно распознает простые фигуры после применения описанных преобразований облака точек.

6.3. Алгоритм классификации с использованием глубокой сверточной нейронной сети

Построим алгоритм классификации двух классов: кругов и квадратов на основе глубокой сверточной нейронной сети (CNN), которая извлекает и анализирует пространственные признаки из изображений. Решение задачи включало следующие этапы:

1. Генерация набора данных с разным расположением, с разным поворотом и различных размеров
2. Предварительная обработка данных: удаление дубликатов и перемешивание изображений
3. Обучение сверточной нейронной сети
4. Анализ полученной точности

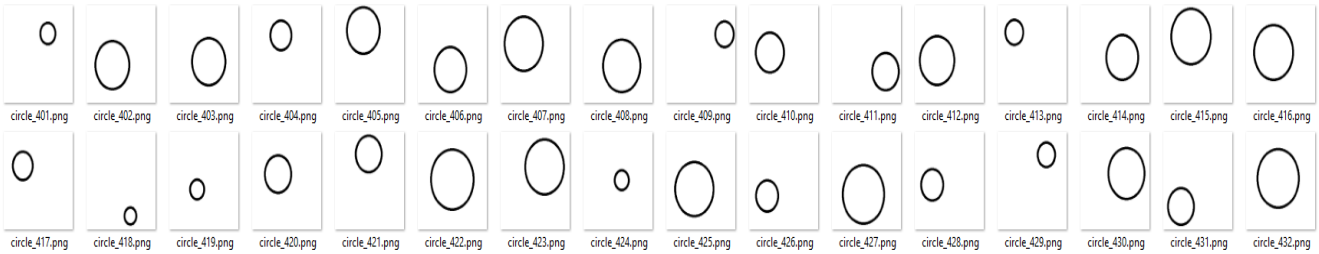
Генерация набора данных

Для начала необходимо сгенерировать набор данных с изображениями из заданного списка (которые мы способны классифицировать). Объекты на изображениях расположены в случайных позициях, имеют разный размер, при этом квадраты дополнительно поворачиваются на случайный угол.

Для генерации кругов со случайным радиусом и случайным положением использовалась функция `cv2.circle`, а для генерации произвольных четырехугольников использовалась функция `cv2.polylines`. Параметры генерации изображений, следующие:

1. Размер изображений 256× 256
2. Радиусы кругов от 20 до 80 пикселей
3. Стороны квадратов от 40 до 120 пикселей
4. Толщина границ фигур 5 пикселей

Таким образом, было сгенерировано 500 изображений кругов и 500 изображений квадратов (рисунок 40):



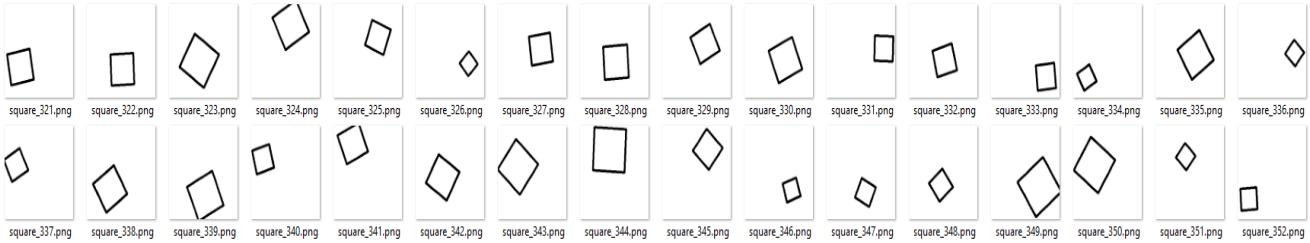


Рисунок. 40. Сгенерированные изображений кругов и квадратов

Предварительная обработка данных

После генерации данных, изображения и их метки были загружены. Метки классов 0 будут соответствовать кругам, а метки классов 1 будут советовать квадратам.

Чтобы улучшить качество обучения модели, исключить переобучение, необходимо предобработать изображения. Поэтому после загрузки данных необходимо убедиться, что среди изображения нет дубликатов и в случае, если таковы имеются, то на этапе предобработки их нужно удалить среди раннее сгенерированных изображений.

Каждое изображение было преобразовано в одномерный массив, где каждый элемент представляет значение интенсивности пикселя (в диапазоне от 0 до 255 для оттенков серого). Так, для выходного одного изображения размером 256×256 – мы имеем массив длиной 65.536. Чтобы найти уникальные строки, будем использовать `np.unique`, который возвращает индексы уникальных строк.

После удаления одинаковых изображений среди раннее сгенерированных (изображений с объектами: круг / квадрат) применяется случайная перестановка изображений (индексов) для равномерного распределения классов в данных. Такой подход используется для того, чтобы обеспечить сбалансированное обучение.

Создание Dataset

Для работы с библиотекой PyTorch был реализован собственный класс CustomShapesDataset, который:

1. Позволяет загружать изображения из указанных директорий
2. Обрабатывает изображения: чтение изображений в оттенках серого, преобразует изображение в формат $1 \times H \times W$
3. Нормализует значения пикселей в отрезок $[0; 1]$

4. Позволяет загружать данные партиями пар <изображение, метка> для обучения

Архитектура CNN

Для классификации была создана сверточная нейронная сеть со следующей архитектурой:

0. Входной слой – принимает изображение размером 150 × 150 пикселей с 3 цветовыми каналами RGB
1. Последовательность нескольких сверточных блоков – используются для извлечения признаков из изображений. Также в конце сверточного блока используется пулинг (`MaxPooling`). Постепенно увеличивается количество каналов, а также уменьшается размер изображения через `MaxPooling`.
   1.1. `nn.Conv2d` – сверточный слой с ядром 3×3 и `padding` = 1 для сохранения пространственного размера. Преобразует тензор с `input_channels` каналами в тензор с `output_channels` каналами.
   1.2. `nn.ReLU` – функция активации `ReLU`, которая добавляет нелинейность в модель, позволяя ей обучаться и распознавать более сложные признаки.
   1.3. `MaxPooling` с ядром 2×2 – используется для уменьшения пространственной размерности, в следствии чего размер изображения уменьшается по высоте и ширине в 2 раза.

Таким образом, последовательность сверточных блоков имеет следующий вид (рис. 37):

```
layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
layers.Conv2D(64, (3, 3), activation='relu'),
layers.Conv2D(128, (3, 3), activation='relu'),
layers.Conv2D(128, (3, 3), activation='relu'),
```

Рисунок. 41. Архитектура CNN

2. Слой Flatten – преобразует тензор в одномерный вектор. Данный слой не имеет обучаемых параметров, и не выполняет линейных преобразований или функций активации. Используется после сверточных слоев.
3. Полносвязные слои
   3.1. Полносвязный слой из 512 нейронов с функцией активации RELU
   3.2. Полносвязный выходной слой с использованием сигмоиды для бинарной классификации. Таким образом, выходным значением нейронной сети мы получим

значение из диапазона [0; 1]. Для бинарной классификации будем использовать пороговое значение 0.5, и тогда все значения ≥ 0.5 будут соответствовать квадрату, а значения ниже 0.5 – кругу.

Таким образом архитектура модели CNN имеет следующий вид (рисунок 42-43):

```
# Создание модели CNN
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(1, activation='sigmoid')])
```

Рисунок. 42. Архитектура построенной CNN

0.	Входное изображение: 3 канала (RGB) размер 150x150		
1.1.	Conv2D	Шаг 1 Ядро 3x3 32 фильтра 148 x 148 x 32	896
1.2.	Активация: ReLU		0
1.3.	MaxPooling2D	Окно: 2x2 Шаг 2 74x74x32	
2.1.	Conv2D	Шаг 1 Ядро 3x3 64 фильтра 72 x 72 x 64	18496
2.2.	Активация: ReLU		0
2.3.	MaxPooling2D	Окно: 2x2 Шаг 2 36x36x64	
3.1.	Conv2D	Шаг 1 Ядро 3x3 128 фильтра 34 x 34 x 128	73856
3.2.	Активация: ReLU		0
3.3.	MaxPooling2D	Окно: 2x2 Шаг 2 17x17x128	
4.1.	Conv2D	Шаг 1 Ядро 3x3 128 фильтра 15 x 15 x 128	147584
4.2.	Активация: ReLU		0
4.3.	MaxPooling2D	Окно: 2x2 Шаг 2 7x7x128	
5	Слой выравнивания Flatten	Выход: 7 * 7 * 128 = 6272	0
6.1.	Полносвязный слой (Dense): 512		3 211 776
6.2.	Активация: ReLU.		
7.1.	Выходной слой (Dense)		513
7.2.	Активация: Sigmoid		

Преобразует трехмерный тензор (7, 7, 128) в одномерный вектор.

Рисунок. 43. Архитектура построенной CNN

Обучение построенной модели CNN

Обучение сверточной нейронной сети (CNN) направлено на оптимизацию параметров, чтобы минимизировать ошибку предсказания и максимизировать точность классификации. Реализация включает этап прямого прохода модели для получения предсказаний.

В качестве функции потерь использовался `CrossEntropyLoss`, а для оптимизации весов модели был выбран алгоритм `Adam`, который адаптивно регулирует скорость обучения для каждого параметра модели, чтобы минимизировать ошибку, возвращаемую функцией потерь.

Модель автоматически запускается на `GPU`, если оно доступно, иначе используется `CPU`. Модель `SimpleCNN` перенесена на выбранное устройство, а `model.parameters()` передаёт все параметры модели в оптимизатор для их обновления при обучении.

Обучение модели проходит в несколько эпох `(epochs)`, в каждой из которых выполняются следующие этапы:

1. Данные загружаются партиями из тренировочного загрузчика `train_loader`: каждая партия изображений и соответствующих меток классов переносится на устройство (`GPU` или `CPU`), заданное параметром `device`.
2. Прямое распространение**:** входные изображения пропускаются через слои модели, в результате чего формируются вероятности принадлежности к классам.
3. Вычисление функции потерь: ошибка вычисляется с использованием функции потерь, сравнивающей предсказания модели и истинные метки классов.
4. Обратное распространение: градиенты функции потерь вычисляются с использованием метода обратного распространения ошибки. На основе этих градиентов оптимизатор обновляет параметры модели.
5. Предсказанные моделью классы сравниваются с истинными метками. Рассчитывается точность — доля верных предсказаний в текущем batch.

Проведение экспериментов

На данном этапе была проведена тренировка модели, результаты которой представлены ниже. Обучение модели проходит в несколько `epochs = 10` (рисунок 44):

```
Epoch: 1/10,Время работы: 27.741 Ошибка: 0.702409,Точность: 0.545000
Epoch: 2/10,Время работы: 28.257 Ошибка: 0.654091,Точность: 0.624000
Epoch: 3/10,Время работы: 28.168 Ошибка: 0.480162,Точность: 0.807000
Epoch: 4/10,Время работы: 28.728 Ошибка: 0.082059,Точность: 0.991000
Epoch: 5/10,Время работы: 27.816 Ошибка: 0.019636,Точность: 1.000000
Epoch: 6/10,Время работы: 28.207 Ошибка: 0.029668,Точность: 0.997000
Epoch: 7/10,Время работы: 28.124 Ошибка: 0.070260,Точность: 0.980000
Epoch: 8/10,Время работы: 28.274 Ошибка: 0.024226,Точность: 0.996000
Epoch: 9/10,Время работы: 28.126 Ошибка: 0.141832,Точность: 0.960000
Epoch: 10/10,Время работы: 28.870 Ошибка: 0.010630,Точность: 0.999000
```

Рисунок. 44. Результаты тренировки модели CNN

Средняя ошибка на обучающей выборке постепенно уменьшалась с каждой эпохой, начиная с 0.702409 и достигая 0.010630 к последней эпохе. Точность модели значительно возросла с 54.5% на первой эпохе до 99.9% на последней эпохе, что демонстрирует хорошую способность модели обучаться на предоставленных данных. Время выполнения одной эпохи в среднем составило ~28 сек.

Далее была проведена классификация изображений для обученной модели на тестовом наборе. На вход подаются изображения, они преобразовываются в черно-белый формат, и размер каждого изображения 256×256 пикселей. После преобразования изображения нормализуются и превращаются в тензор. Модель переводится в режим оценки `model.eval()`. На вход модели подаются изображения и для каждого из них определяется вероятность принадлежности к классам.

На основе обработки изображений и работы модели были получены предсказания для двух классов: `"Circle"` (круг) и `"Square"` (квадрат). Результаты классификации для каждого изображения визуализированы на рисунок 45:

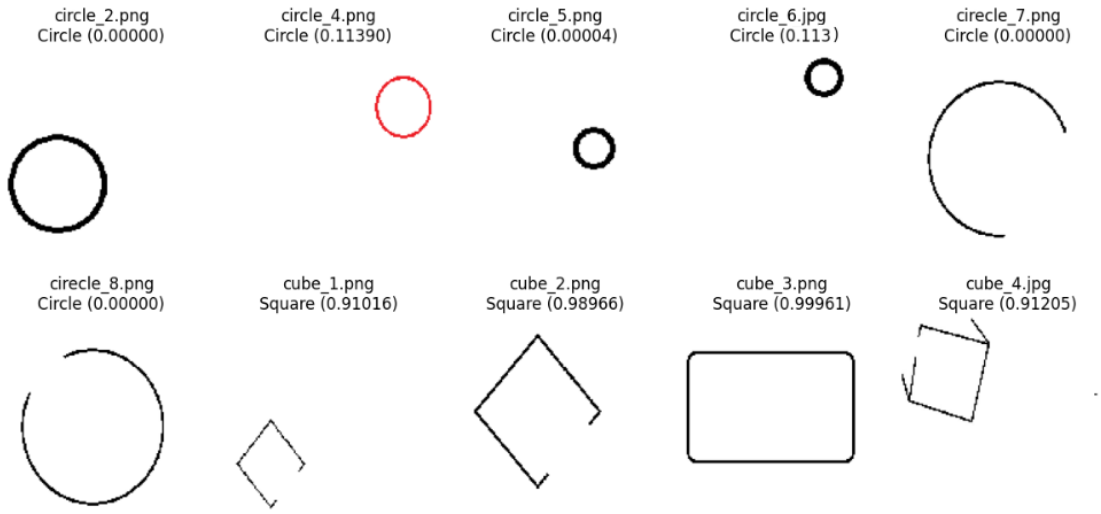


Рисунок. 45. Результаты CNN на тестовом наборе данных

Модель хорошо предсказывает класс для большинства изображений с высокой вероятностью для большинства изображений, что свидетельствует о ее способности распознавать объекты обоих классов — `Circle` и `Square`.

Вывод

В главе был изучен и применен алгоритм кластеризации DBSCAN для реальной задачи кластеризации точек объектов, полученных сканированием с фотографии. Результаты показали корректную работу метода для поставленной задачи. Эксперименты на двух примерах также подтвердили эффективность алгоритма для кластеризации точек.

Также для классификации тестовых примеров был применен метод, который основывается на сравнении расстояний, углов и свертки. Были проведены эксперименты и результаты также показали корректную работу выбранного метода для классификации объектов, основываясь на проекциях облака точек на выбранную ось. Для уточнения результатов классификации была разработана CNN, которая также показала хорошие результаты на тестовых данных, и успешно классифицировала объекты из заданного множества.

Глава 7. Практическое применение: реконструкция помещения

Задача восстановления 3D-модели сцены в виде полигональной сетки заключается в преобразовании двумерного изображения или набора изображений в трехмерное представление, описываемое вершинами, рёбрами и гранями, которые формируют поверхность объекта.

Одним из способов решения данной задачи являются методы, использующие оценку карты глубины изображения. Далее, используя полученную карту глубину, мы можем вычислить координаты трехмерного облака точек, вычислить нормали к полигонам и реконструировать поверхность объектов, создав полигональную сетку. Описанный метод позволяет понять пространственное расположение объектов и их форму в трехмерном пространстве.

Если входное изображение имеет размер $w \times h$, то карта глубины – это матрица $D = (D_{x,y}) \in R^{w \times h}$. Каждый ее элемент $D_{x,y}$ в x столбце и y строке – это вещественное число z $\in R$ (спрогнозированная глубина в точке x, y). Таким образом, методы оценки карты глубины работают с каждый пикселем изображения.

Облако точек в 3D-пространстве – это множество точек с координатами (x, y, z). Более того, каждая точка соответствует пикселю на изображении, и поэтому сохраняет его цветовую информацию в формате (R, G, B).

Размер массива облака точек можно сократить, выбирая не все элементы из карты глубины $D = (D_{x,y}) \in R^{w \times h}$, а только такие D_{x^*,y^*}, которые соответствуют границам объектов на изображении (например границы выделены методом Canny, который был описан раннее в главе 5 п. «Применение метода `Canny`») . Сокращение размера массива трехмерного облака точек позволяет сократить затраты на хранение массива, а также снизить требования к вычислительным ресурсам (т.к. плотность облака точек значительно меньше по равнению, если в массив записывать все точки исходного изображения). Такое разреженное облако точек упрощает кластеризацию и в дальнейшем классификацию объектов на сцене.

Если облако точек в трёхмерном пространстве достаточно плотное, то на его основе можно восстановить поверхность, сформировав полигональную сетку. Чаще всего используют треугольную полигональную сетку, состоящую из множества соединённых вершин, формирующих трёхмерные треугольники. Чтобы сформировать полигональную сетку,

необходимо задать порядок соединения полигонов, то есть определить последовательность обхода вершин. А также необходимо указать нормали для каждого полигона, чтобы правильно определить их ориентацию в пространстве.

Одним из способов решения задачи реконструкции поверхности является алгоритм Пуассона. Этот алгоритм наиболее устойчив к шуму и подходит для сложных объектов, например, как в нашем случае, когда фотографий немного, и отсканированное облако точек содержит шумы. Также для решения данной задачи существуют и другие методы, такие как:

1. алгоритм альфа-форма: простой в реализации, подходящий для примитивных форм;
2. метод шарового вращения: работает быстро и эффективно.

В рамках данной магистерской диссертации для решения задачи восстановления 3D-модели будет использоваться одно изображение. Сначала будет получена оценка карты глубины, а затем на её основе с помощью алгоритма Пуассона будет выполнена реконструкция трёхмерной модели.

7.1. Методы решения

На вход задачи подаётся единственное изображение $I = (I^{k=3}{}_{x,y})$ в формате RGB, размера $w \times h$. $(0 \leq x < w; 0 \leq y < h)$

Оценка карты глубины

Оценка карты глубины на основе одного RGB-изображения называется монокулярной оценкой глубины. Результатом такой оценки является карта глубины $D = (D_{x,y}) \in R^{w \times h}$ — матрица, в которой каждый элемент соответствует предсказанному значению глубины z = $D_{x,y}$ для соответствующего пикселя (x, y) исходного изображения.

Для получения оценки карты глубины $\tilde{z} = \widetilde{D} = (\widetilde{D}_{x,y})$ мы будем использовать модель глубокого обучения GLPN `(Global-Local Path Network)`, основанную на сверточных нейронных сетях (CNN). При этом отключим вычисление градиентов, чтобы ускорить процесс предсказания и уменьшить количество используемой памяти. Модель предварительно обучена на наборе данных «NYU Depth V2», который включает в себя сцены различных помещений. Но прежде, чем подать изображение на вход модели, его необходимо предобработать:

1. привести изображение к размеру, кратному 32
2. преобразовать в многомерный массив, тензор, и нормализовать значения пикселей из диапазона [0; 255] в диапазон [-1, 1].

Набор данных «NYU Depth V2»

Набор данных «NYU Depth V2» разработан Нью-Йоркским университетом (NYU), и включает 1449 пар изображений RGB-глубина. Изображения имеют разрешение 640 × 480 пикселей и сохранены в формате JPEG. Данные ограничены сценами внутри помещений, что может быть недостатком для задач, связанных с уличными сценами, но этого достаточно для целей, поставленных в данной работе. Сбор информации осуществлялся с помощью устройства `Microsoft Kinect`, которое объединяет в себе функции камеры, датчика глубины и микрофона.

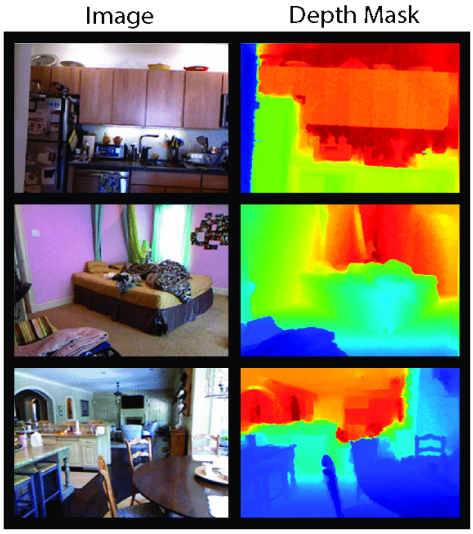


Рисунок. 46. Пример данных в «NYU Depth V2»

Постобработка предсказанной карты глубины

После того, как модель предсказала карту глубины $\tilde{z} = \widetilde{D} = (\widetilde{D}_{x,y})$ для изображения $I = (I^{k=3}{}_{x,y})$ на входе, необходимо выполнить постобработку полученной карты глубины. Так, мы улучшим качество и подготовим карту глубины для дальнейшего использования. Выполним следующие преобразования:

1. так как модель возвращает тензор размерностью (1, h, w), необходимо удалить лишнее измерение, чтобы получить размерность (h, w). В дальнейшем этот упростит обработку.
2. если тензор находится на GPU, то он перемещается на CPU и преобразуется в NumPy массива для удобства работы.
3. значения глубины $\widetilde{D}_{x,y}$ умножаются на 1000.0, преобразуя глубину из м. в мм.
4. так как модель при обработке изображения добавляет padding, необходим удалить границы, по 16 пикселей с каждой стороны. Чтобы изображение соответствовало полученным картам глубины по размеру – необходимо и изображение обрезать с каждой стороны по 16 пикселей.
5. карта глубины $\tilde{z} = \widetilde{D} = (\widetilde{D}_{x,y})$ нормализуется в диапазон [0, 255] так, чтобы максимальное значение цвета пикселя стало 255. Результат преобразуется в тип uint8 (целые числа от 0 до 255).

6. создается RGB-D изображение, которое объединяет цвет и глубину. Таким образом, карта глубины и цветное изображение преобразуются в объекты Image из библиотеки Open3D. После постобработки мы имеем:

1. карту глубины $\acute{z} = \acute{D} = \left(\acute{D}_{x,y}\right)$
2. изображение $\acute{I} = \left(\acute{I}^{k=3}{}_{x,y}\right)$, где $0 \leq x < \acute{w} = w - 16; 0 \leq y < \acute{h} = h - 16$
3. RGB-D изображение I_z - объект, содержащий два атрибута: color, depth).

Облако точек в трехмерном пространстве

Теперь на основе RGBD-изображения I_z мы можем получить 3D-облако точек. Зная точные параметры камеры, мы их можем указать, и таким образом, получим облако точек, где расстояния между точками соответствуют реальным значениям. Параметры камеры задаются с помощью экземпляра класса `PinholeCameraIntrinsic` из библиотеки Open3D. Класс `PinholeCameraIntrinsic` имеет метод `set_intrinsics` для задания внутренних параметров камеры. После чего этот экземпляр класса передаётся в функцию `o3d.geometry.PointCloud.create_from_rgbd_image` вместе с RGBD-изображением I_z, и на выходе мы получаем облако точек:

$$\mathrm{P} = \{p_i\} = \{(x_i, y_i, z_i):\ i = \overline{0, n-1}\}$$

Для генерации полигональной сетки `(polygonal mesh)`, воспользуемся алгоритмом «реконструкции поверхности Пуассона». Алгоритм Пуассона был выбран для реконструкции поверхности, поскольку он наиболее устойчив к шуму и хорошо подходит для работы со сложными объектами, обеспечивая гладкую и детализированную поверхность. Другие алгоритмы, а именно алгоритм альфа-форма `(Ball Pivoting)` и метод шарового вращения `(Alpha Shapes)`, были кратко описаны во введении к этой главе.

Для начала необходимо удалить выбросы из облака точек. Для этого необходимо указать количество ближайших соседей k, которые будут анализироваться для фиксированной точки $p_i \in P = \{(x_i, y_i, z_i):\ i = \overline{0, n-1}\}$, а также пороговое значение стандартного отклонения σ. Для удаления выбросов из облака точек, необходимо выполнить следующее:

1. Для каждой точки $p_i \in P$ вычислить среднее значение расстояний до ее k ближайших соседей d_i. Тогда мы получим список средних расстояний для этих соседей:

$$\{\mathrm{d}_0, \dots, d_{k-1}\}$$

2. Для каждой группы k-ближайших соседей вычислим среднее значение $mean = \frac{1}{k}\sum_{j=0}^{k} d_j$, а также стандартное отклонение $std = \sqrt{\frac{1}{k}\sum_{j=0}^{k}(d_j - mean)^2}$, которое показывает, насколько сильно расстояние d_j отклоняется от среднего значения $mean$.
3. Теперь для каждой группы ближайших соседей вычисляется пороговое значение

$$\mathrm{C} = mean + \sigma \cdot std.$$

4. Если для точки p_i $(i = \overline{1,k})$ среднее расстояние d_i относительно ее соседей превышает пороговое значение C, то точка p_i считается выбросом, и она удаляется из мн-ва P

Облако точек в трехмерном пространстве для границы объектов

Границы объектов получим с помощью метода Canny, который в зависимости от яркости пикселя определяет, принадлежит ли пиксель границе или нет. Т.к. алгоритм Canny работает с одноканальными изображениями, то предварительно необходимо конвертировать изображение в оттенки серого. Результат алгоритма - изображение $I_{\text{conture}} = (I_{\text{conture}}{}^{k=1}{}_{x,y})$ $0 \le x < w\,; 0 \le y < h$ со значениями 0, что соответствует черному цвету, либо 255, что соответствует белому цвету, а именно границе:

$$I_{\text{conture}}(x,y) = \begin{cases} 255, (x,y) \text{ принадлежит контуру} \\ 0, \text{иначе} \end{cases}$$

Маска контура применяется к карте глубины $\acute{z} = \acute{D} = (\acute{D}_{x,y})$, сохраняя в $\acute{D}$ только те элементы $\acute{D}_{x,y}$, где $I_{\text{conture}}(x,y) = 255$. И далее, аналогично предыдущему пункту из полученной карты глубины создается трехмерное облако точек сцены.

Нормали к полигонам mesh-сетки

После удаления шумовых точек из множества $P = \{(x_i, y_i, z_i)\colon i = \overline{0, n-1}\}$, необходимо вычислить нормали для каждой точки $p_i \in \tilde{P} \subseteq P = \{(x_i, y_i, z_i)\colon i = \overline{0, n-1}\}$, где $\tilde{P}$ — облако точек после удаления выбросов. Нормаль вычисляется для локальной плоскости, которая наилучшим образом аппроксимирует $\tilde{k}$ ближайших соседей точки p_i $(i = \overline{0, n-1})$. Эта нормаль и будет перпендикулярна данной плоскости, и таким образом, полученная нормаль характеризует ориентацию поверхности в окрестности ε для точки p_i.

Так, мы получим множество нормалей N = $\{n_i = (n_{i,x}, n_{i,y}, n_{i.z})\colon\ i = \overline{0, n-1}\}$, где каждая нормаль n_i соответствует одной точке p_i в облаке $\tilde{P}$.

При вычислении нормалей, их направление определялось локальной геометрией $\tilde{k}$ ближайших соседей точки p_i $(i = \overline{0, n-1})$. Но направление всех нормалей $n_i \epsilon\ N = \{n_i = (n_{i,x}, n_{i,y}, n_{i.z})\colon i = \overline{0, n-1}\}$должно быть согласовано все нормали должны быть направлены либо внутрь, либо наружу поверхности. Иначе были бы проблемы при визуализации и реконструкции поверхности, а именно при реконструированная поверхность была бы некорректной и имела или артефакты. На вход подается опорное направление, например, вектор $Z = (0, 0, 1)$. И для каждой точки $p_i \in \tilde{P}$ вычисляется скалярное произведение нормали n_i и опорного направления Z:

$$multi = (n_i; Z)$$

Если результат скалярного произведения отрицательный ($multi$ <0), вектор нормали для точки p_i инвертируется (умножается на -1). Таким образом, после выполнения данной процедуры все нормали $n_i\ \epsilon\ N$ будут направлены в одну сторону ($i = \overline{0, n-1}$).

Алгоритм восстановления поверхности Пуассона

Теперь мы можем реконструировать поверхность на основе облака точек $\tilde{P} \subseteq P = \{(x_i, y_i, z_i)\colon\ i = \overline{0, n-1}\}$, и для этого применим алгоритм Пуассона. Рассмотрим, как работает алгоритм.

Пусть $\Omega \subset R^3$ – ограниченная замкнутая область с непустой внутренностью int Ω и гладкой границей $\partial\Omega$. Индикаторная функция $\chi(p)$ – скалярная функция, которая определяет находится ли точка $p(x, y, z)$ внутри или снаружи некоторой области $\Omega \subset R^3$:

$$\forall p_i \in \tilde{P} \subseteq P = \{(x_i, y_i, z_i)\colon\ i = \overline{0, n-1}\}\colon$$

$$\chi(p_i) = \begin{cases} 1, p_i \in int\Omega \\ 0, p_i \notin R^3 \setminus \Omega \end{cases}$$

Существует взаимосвязь между индикаторной функцией $\chi(p_i)$ и нормалями $n_i \epsilon\ N = \{n_i = (n_{i,x}, n_{i,y}, n_{i.z})\colon\ i = \overline{0, n-1}\}$, которые соответствуют каждой точке $p_i \in \tilde{P} \subseteq P = \{(x_i, y_i, z_i)\colon\ i = \overline{0, n-1}\}$. Известно следующее выражение:

$$\nabla\chi = \vec{V}, \text{где } \vec{V} - \text{это векторное поле нормалей}$$

Так, например, для точки $\tilde{p}$, удаленной от объекта, $\nabla\chi = 0$, т.к. в ее окрестности $\forall p^* \in (p_k, p_{k+1}, \dots)\colon \chi(p^*) = 0$. Если же точка $\tilde{p} \in int\Omega$, то $\forall p^* \in (p_k, p_{k+1}, \dots)\colon \chi(p^*) = 1$, и

соответственно $\nabla\chi = 0$. Но вдоль границы $\partial\Omega$ градиент индикаторной функции $\nabla\chi$ совпадает с направленной внутрь нормалью $\tilde{n}$.

Если с обоих сторон уравнения применить оператор дивергенции, то получим уравнение Пуассона, которое уже известно, как решать, и таким образом, мы сможем восстановить неизвестную индикаторную функцию $\chi(p)$:

$$\nabla\nabla\chi = \Delta\chi = \nabla\vec{V}$$

$$\Delta\chi = \nabla\vec{V} \quad (1)$$

Уравнение Пуассона (1) является ключевым для алгоритма реконструкции поверхности. Оно связывает индикаторную функцию $\chi(p)$ с векторным полем нормалей $\vec{V}$, и решение этого уравнения позволяет восстановить $\chi(p)$, что, в свою очередь, позволяет определить, какие точки пространства находятся внутри объекта ($\chi(p) = 1$).

Примечание: соотношение между градиентом индикаторной функции и векторным полем нормалей

На вход мы имеем облако точек $\tilde{P} \subseteq P = \{(x_i, y_i, z_i)\colon\ i = \overline{0, n-1}\}$ и для каждой точки нормаль $\overrightarrow{n_i} \epsilon\, N$, которая лежит на границе ∂M восстанавливаемой неизвестной замкнутой модели М или вблизи этой границы.

Т.к. индикаторная функция χ кусочно-постоянная, то вычисление ее градиента привело бы к получения векторного поля с неограниченными значениями на границе ∂M. Применим сглаживающий фильтр, и тогда будем вычислять градиент для сглаженной индикаторной функции.

Лемма (Kazhdan, Bolitho, Hoppe, 2006). Пусть дано тело М с границей ∂M, χ_M – индикаторная функция М. $\overrightarrow{N_{\partial M}}(p)$ – нормаль к поверхности, направленная внутрь, в точке $p \epsilon\, \partial M$. Сглаживающий фильтр $\tilde{F}_p(q) = F(q-p)$, переводящий в точку p**.** Тогда градиент сглаженной индикаторной функции равен векторному полю, полученному сглаживанием поля нормалей поверхности:

$$\nabla\left(\chi\tilde{F}\right)|_{q_0} = \int_{\partial M} \tilde{F}(q_0 - p\,)\overrightarrow{N_{\partial M}}(p)dp$$

Доказательство данной леммы содержится в работе [11] (раздел 3 "Our Poisson reconstruction approach", С. 3-4).

Далее аппроксимируем интеграл, используя входной набор точек из облака. Разобьём границу ∂M на участки $\mathcal{M}_{p^*} \subset \partial M$, где в каждом участке $\mathcal{M}_{p^*}$ лежит единственная точка p^* из множества $\tilde{P}$. Таким образом, аппроксимируем криволинейный интеграл по границе ∂M суммой по всем таким участкам $\mathcal{M}_{p^*}$, и получим следующее выражение:

$$\nabla(\chi\tilde{F})|_q = \int_{\partial M} \tilde{F}(q - p)\overrightarrow{N_{\partial M}}(p)dp = \sum_{p\epsilon\tilde{P}} \int_{\mathcal{M}_p} \tilde{F}(q - \tilde{p})\overrightarrow{N_{\partial M}}(\tilde{p})dp \approx \sum_{p\epsilon\tilde{P}} |\mathcal{M}_p| \cdot \tilde{F}(q - p) \cdot \overrightarrow{N_{\partial M}}|_p \equiv \vec{V}(q)$$

$$\nabla\tilde{\chi} = \vec{V}$$

7.2. Описание программной реализации

Программная реализация реконструкции помещения с применением алгоритма Пуассона для восстановления поверхности была разработана на языке Python в среде Jupyter Notebook. Полная реализация перечисленных ниже функций представлена в [Приложение Н].

При реализации были использованы следующие вспомогательные функции:

1. `def load_and_prepare_image(image_path)` — функция загружает изображение по указанному пути `image_path` и изменяет его размер таким образом, чтобы высота и ширина изображения были кратны 32. Функция возвращает предобработанное изображение.
2. `def predict_depth(model, inputs)` — функция выполняет предсказание карты глубины для изображения, которое было загружено раннее. При этом высота и ширина изображены должны быть кратны 32. На выходе функция возвращает предсказанную глубину, которая содержится в атрибуте `predicted_depth` объекта `outputs`.
3. `def remove_borders (depth_map, image, pad = 16)` — функция уменьшает размерность `map depth_map` удаляя лишнее измерение для тензора данной карты глубины. А также внутри выполняется обрезка по границам с шагом pad = 16 для карты глубины и для самого изображения. Функция возвращает постобработанную карту глубины и изображение.
4. `def apply_contour_mask (depth_map, image, contour_mask)` — функция для удаления элементов матрицы карты глубины, которые не принадлежат границе, выделенной методом cv2.Canny. На вход подается предсказанная карта глубины для всего

изображения, само изображение, и бинарная маска контура. Функция возвращает карту глубины с нулевыми элементами, лежащими не на границе, а также соответствующее изображение для полученной карты глубины.

5. `def visualize_results (image, depth_map)` – функция для визуализации результата предсказанной карты глубины на предыдущих шагах метода. На вход принимает постобработанное изображение и саму карту глубины.
6. `def create_point_cloud(image, depth_map)` – функция создает трехмерное облако точек `point_cloud` на основе RGB изображения image и карты глубины depth_map, выполняется очистка от шумовых точек и вычисляются нормали для каждой точки из `point_cloud` (нормали к полигонам, построенным на основе k ближайших соседей – см. п. 7.1. «Нормали к полигонам `mesh`-сетки»).

   Функция возвращает `open3d.geometry.PointCloud` – класс облака точек, который включает в себя поля для хранения координат и цвета точек (цвета соответствуют пикселю исходного изображения), а также вычисленные нормали.
7. `def reconstruct_surface(point_cloud)` – функция выполняет реконструкцию поверхности методом Пуассона (см п. 7.1. «Алгоритм восстановления поверхности Пуассона») на основе `point_cloud`.

   Функция возвращает объект `o3d.geometry.TriangleMesh`, представляющий собой 3D-mesh модели.
8. `save_and_visualize_mesh(mesh, point_cloud, output_path)` – функция сохраняет 3D-mesh в файл и визуализирует `point_cloud`

Используемые тип данных:

1. Модель `GLPN (Global-Local Path Network)`:
   1.1. `GLPNImageProcessor` – класс для выполнения предобработки изображений, которые в дальнейшем поступают на вход модели GLPN
   1.2. `GLPNForDepthEstimation` – класс модели, используемый для получения предсказаания карты глубины исходного изображения (depth_map). Возвращет предсказание в виде тензора, который далее необходимо преобразовать в изображение для дальнейшего применения.

2. `PIL.Image.Image` – класс для загрузки и обработки изображения
3. `open3d.geometry.Image` – класс для хранения изображения в формате `Open3D`
4. `open3d.geometry.RGBDImage` – класс для хранения карбы глубины и исходного изображения
5. `open3d.camera.PinholeCameraIntrinsic` – класс для хранения описаний параметров камеры
6. `open3d.geometry.PointCloud` – класс для хранения облаков точек в 3D пространстве

7.3. Результаты

Проанализируем результаты применения алгоритма Пуассона для реконструкции поверхности на реальных фотографиях помещений и комнат: для этого визуализируем облако точек и полученные реконструированные 3D-модели, а также отфильтрованное облако точек и реконструированные модели для них, оставляя только точки, соответствующие границам объектов.

Рассмотрим следующие примеры изображений комнаты для реконструкции помещений (рисунок 47 - 49):

Рисунок. 47. Пример 1: спальня

Рисунок. 48. Пример 2: гостиная

Рисунок. 49. Пример 3: кухня

Применим функцию 1-3 из п. 7.2. «Описание программной реализации», и на выходе получим предсказанную и постобработанную карту глубины для каждого из изображения (рисунок. 50 - 52):

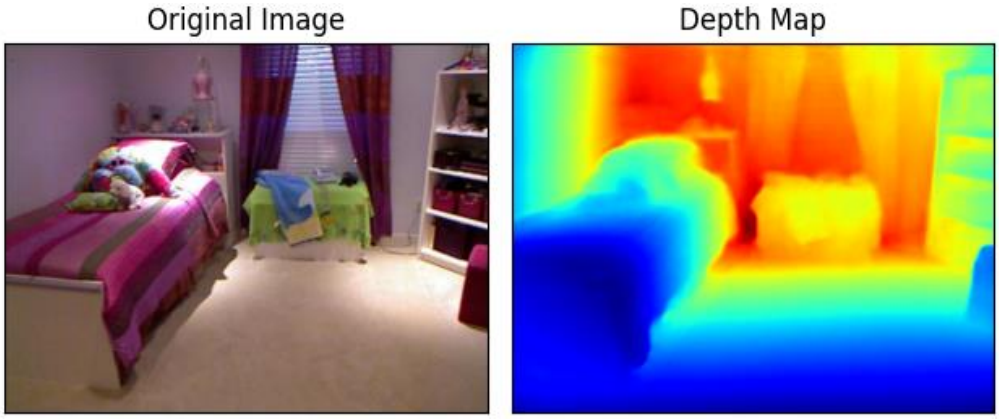


Рисунок. 50. Пример 1: карта глубины (спальня)

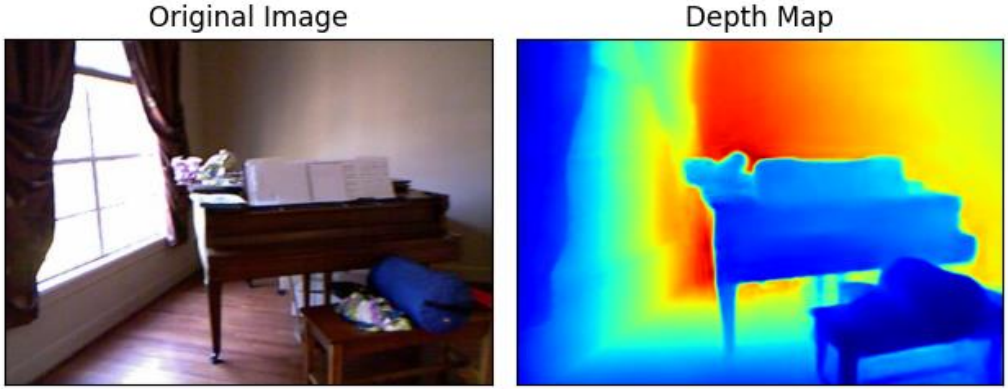


Рисунок. 51. Пример 2: карта глубины (гостиная)

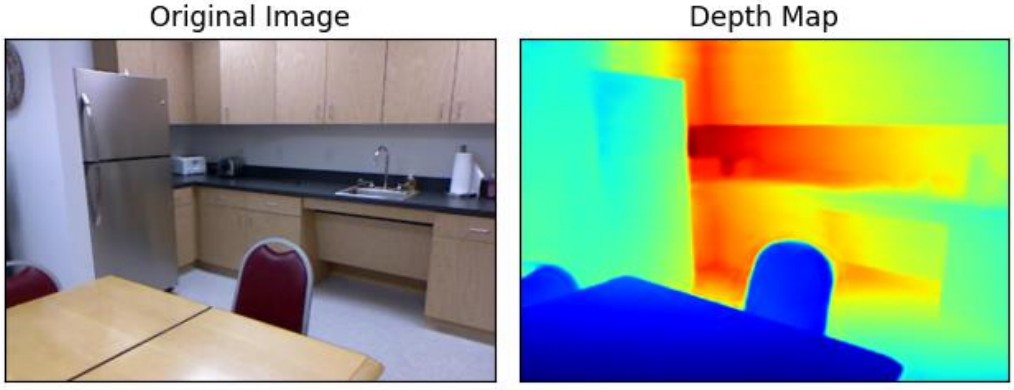


Рисунок. 52. Пример 3: карта глубины (кухня)

Таким образом, для каждого изображения мы имеем карту глубины, в которой каждому пикселю соответствует его расстояние до объекта – элемент матрицы карты глубины. Применим функцию 6 из п. 7.2. «Описание программной реализации», и на выходе получим облако точек в трехмерном пространстве. Для сравнения также получим облако точек в трехмерном пространстве из карты глубины, в которой ненулевые значения имеют только элементы, лежащие на контуре объектов сцены (рисунок. 53-55):

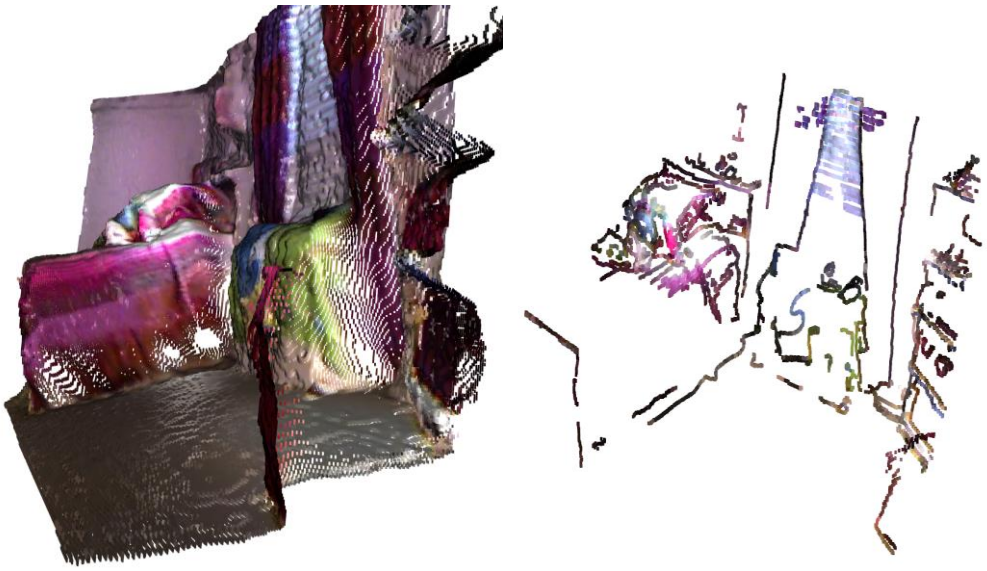


Рисунок. 53. Пример1: трехмерное облако точек и облако граничных точек (спальня)

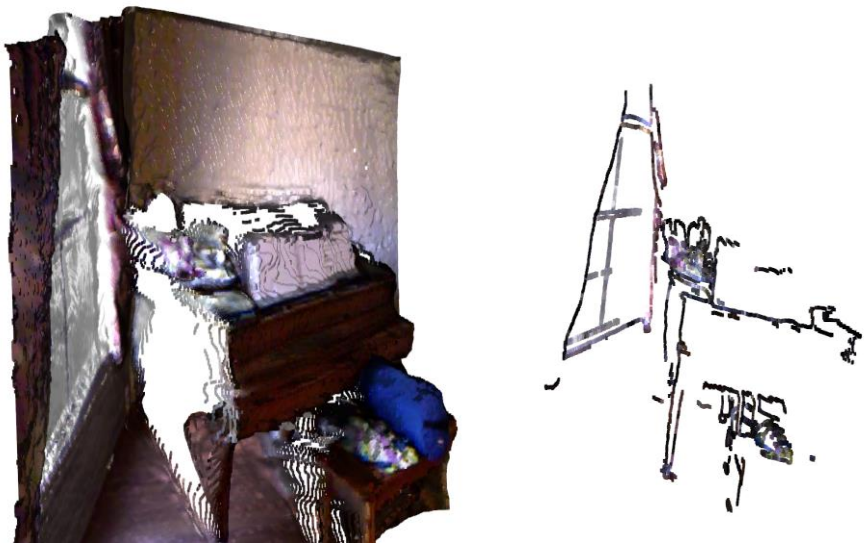


Рисунок. 54. Пример2: трехмерное облако точек и облако граничных точек (гостиная)

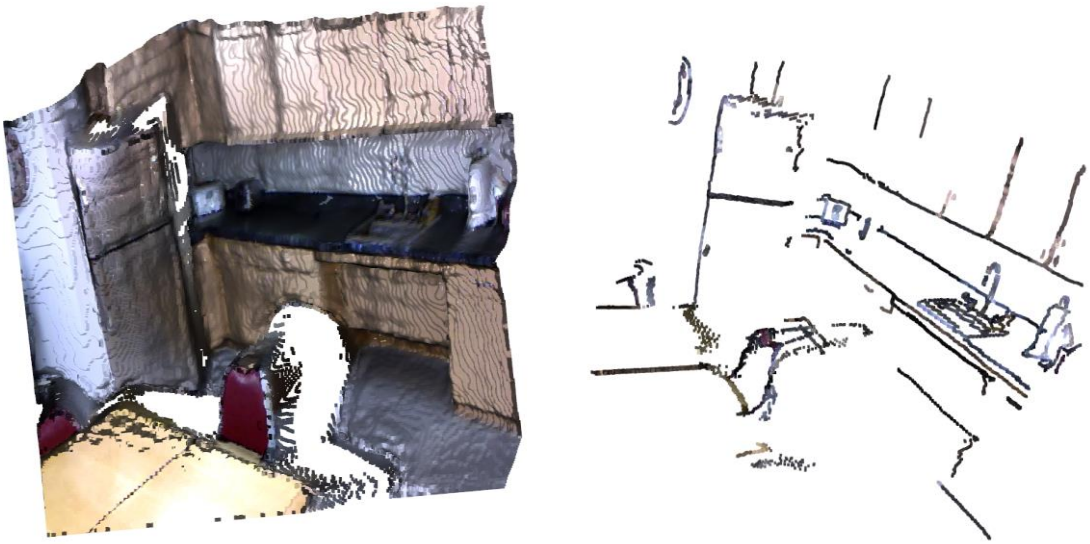


Рисунок. 55. Пример3: трехмерное облако точек и облако граничных точек (кухня)

Теперь применим к облаку точек алгоритм реконструкции поверхности Пуассона (функция 7 из п. 7.2. «Описание программной реализации»). Таким образом, получим 3D модель, сохраненную в .obj формате (рисунок 56-58):

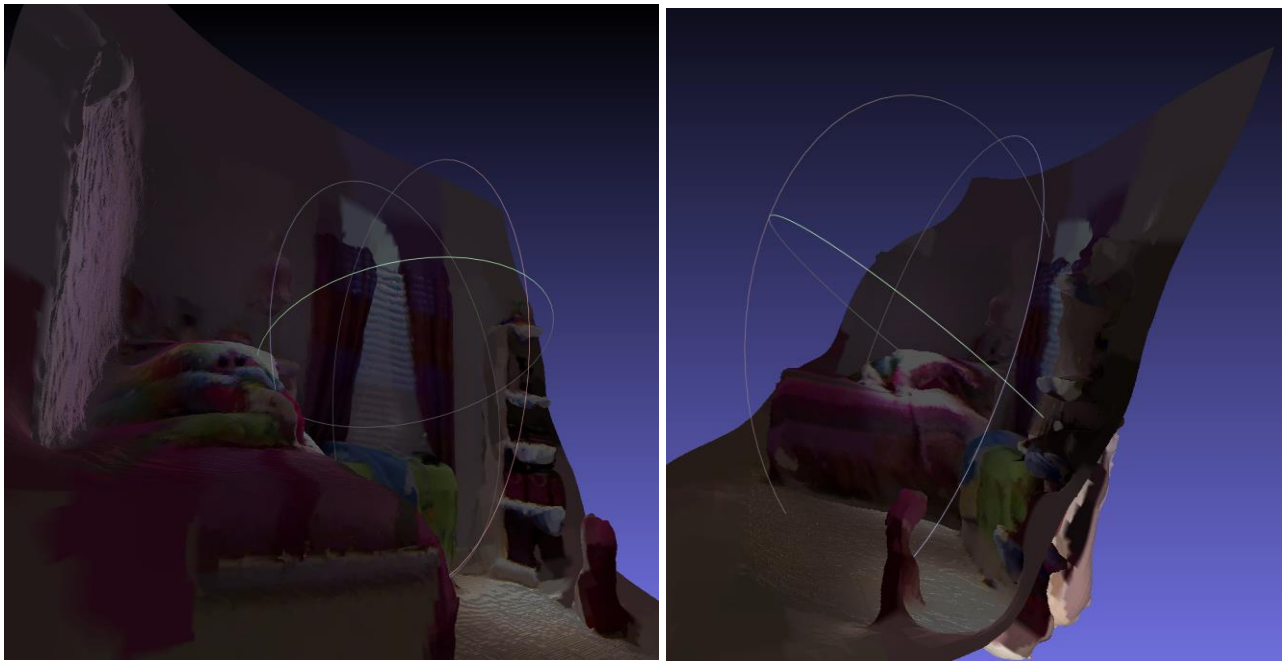


Рисунок. 56. Пример1: восстановленная поверхность методом Пуассона на основе 3D облака точек (спальня)

Рисунок. 57. Пример2: восстановленная поверхность методом Пуассона на основе 3D облака точек (гостиная)

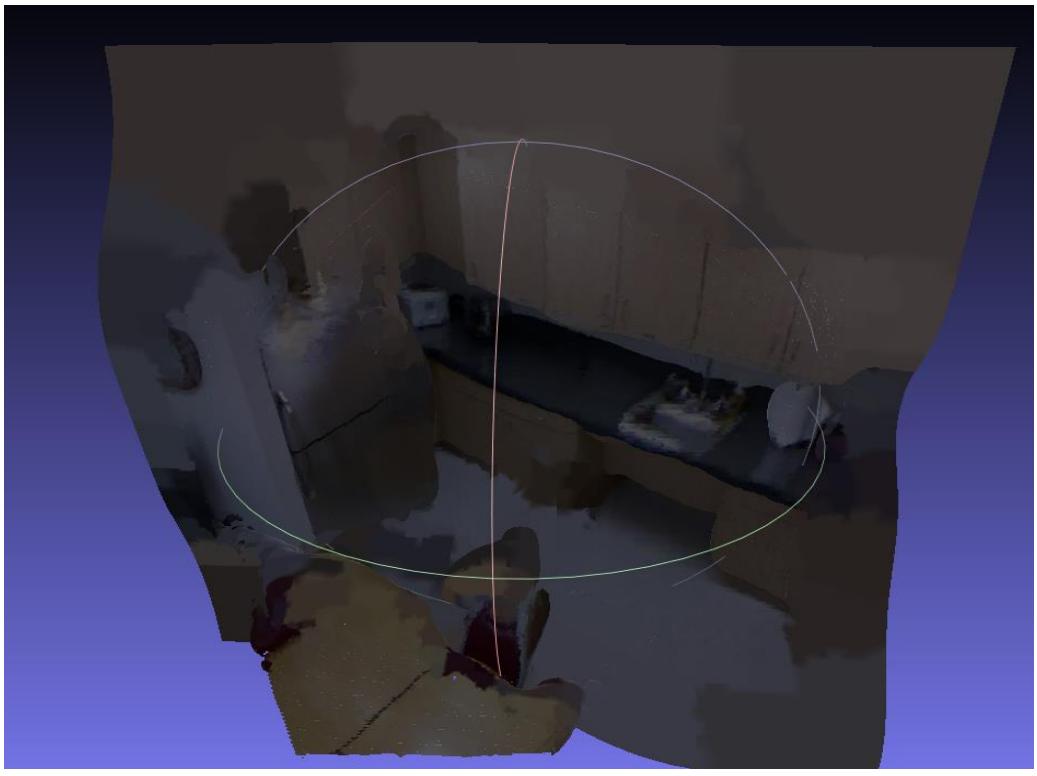


Рисунок. 58. Пример3: восстановленная поверхность методом Пуассона на основе 3D облака точек (кухня)

Как видно из приведенных результатов запуска программы, получается достаточно качественная реконструированная сцена. Наиболее корректно получилось восстановить методом Пуассона поверхность тех объектов, которые видно с разных ракурсов, и которые имеют достаточное количество точек в трёхмерном облаке.

Также можем заметить, что трехмерное облако точек, построенное на основе карты глубины с ненулевыми значениями только на границе, в дальнейшем могло бы упростить кластеризацию и классификацию объектов, находящихся на сцене. Например, на рис 47. По облаку граничных точек мы можем четко различить холодильник, кран, шкафы и т.д. аналогично на других раннее приведенных рисунках (45-46).

Таким образом, плотное облако точек подходит для задачи реконструкции поверхности методом Пуассона. А облако граничных точек мы можем использовать для разбиения на кластеры и классификации, т.к. объекты легко различимы и образуют небольшие группы из подмножества облака точек. Провести кластеризацию, например, можно алгоритмом DBSCAN, реализация которого была приведена ранее в Главе 6 «Кластеризация и классификация массива точек после применения метода оптического потока»

Вывод

В данной главе был изучен и применен алгоритм восстановления поверхности методом Пуассона с использованием одной фотографии помещения. Для применения этого алгоритма предварительно необходимо: получить карту глубины для исходного изображения, из карты глубины получить облако точек и далее вычислить нормали к мэш-полигонам, из которых и будет состоять трехмерная восстановленная модель. Таким образом, также был рассмотрен новый способ получения трехмерного облака точек из изображений.

Результаты запуска программной реализации показали корректную работу метода восстановления поверхности: на выходе алгоритма мы получили трехмерную модель сцены помещения из одного изображения. Было проведено три эксперимента с разными изображениями: для них была получена карта глубины, трёхмерное облако точек и восстановленная 3D-модель сцены.

Также была рассмотрена реализация получения трехмерного облака точек из карты глубины, в которой ненулевые значения имеют только те элементы, которые принадлежат пикселям контура объектов исходного изображения. Из приведенных результатов на рис.

45-47 можно заметить, что такой массив точек имеет меньшую плотность в отличие от метода, учитывающего все значения из карты глубины. Таким образом, облако граничных точек в дальнейшем можно разбить на кластеры и проклассифицировать, чтобы выделить отдельные объекты сцены. Однако, методом Пуассона не получится восстановить поверхность такого разреженного облака точек, т.к. расстояние между точками достаточно большое, и поэтому модель получится низкополигональная. Если же разбить трехмерное облако граничных точек на кластеры и применить метод Пуассона к каждому кластеру, то в итоге можно получить 3D-модели для каждого объекта на сцене.

Заключение

В данной работе были изучены технологии виртуальной и дополненной реальности VR/AR для реконструкции помещений на основе фотографических данных. В ходе курсовой работы были рассмотрены и применены различные методы для отслеживания объектов по фотографиям, метод Пуассона для реконструкции поверхности, а также изучена и выполнена программная реализация для технологии отслеживания плоскостей в дополненной реальности (AR).

В результате проведенных исследований и работы были получены следующие основные результаты:

1. Виртуальная и дополненная реальность:
   1.1. Изучена технология AR PlaneTracking, реализована и протестирована на ОС Android для распознавания горизонтальных и вертикальных поверхностей в режиме реального времени при помощи камеры смартфона.
   1.2. Выполнена программная реализация визуализации части комнаты (стены, пол и потолок) на основе фотографии из центра комнаты
2. Распознавание (сканирование) комнаты по серии фотоданных:
   2.1. В рамках работы был проведен анализ методов распознавания объектов на изображениях, и на основе этого анализа была выполнена программная реализация для дальнейшего сравнения их работы
   2.2. Проведен ряд экспериментов для сравнения методов распознавания объектов (плотный поток Фарнебака, стереоректификация с использованием stereoBM):
   В результате сравнения был сделан вывод, что при неизвестных параметрах камеры на практике лучше использовать метод плотного потока, чем применение стереоректификация с использованием stereoBM
   2.3. Выполнена программная реализация метода плотного потока для решения задачи распознавания объектов на двух стереоизображениях. В результате чего удалось корректно распознать объекты с фотографии и визуализировать их в ПО Blender.
   2.4. Проведена кластеризация распознанных объектов с использованием алгоритма DBSCAN для последующей классификации отсканированных объектов. Алгоритм работает корректно и на тестовом примере правильно класстеризовал множество объектов, состоящих из облака точек в трехмерном пространстве.

2.5. Выполнена программная реализация классификации объектов для каждого кластера, распознанных объектов на сцене:

2.5.1. Использовался метод, основанный на геометрических характеристиках, включая расстояние до центра и угловую координату точки относительно центра, а также на результатах свёртки вектора расстояний.

2.5.2. В качестве дополнительного метода классификации была построена сверточная нейронная сеть (CNN), обеспечивающая улучшение точности классификации за счет извлечения признаков. Оба подхода эффективно классифицируют объекты. Эти методы можно использовать вместе для уточнения результатов, комбинируя геометрические признаки с другими признаками, извлекаемыми с помощью CNN.

3. Практическое применение. Реконструкция помещения, алгоритм Пуассона:

3.1. Изучены и рассмотрены различные методы для реконструкции сцены (восстановления поверхности) по трехмерному облаку точек. В качестве одного из таких методов был выбран метод Пуассона, т.к. этот алгоритм наиболее устойчив к шуму, что актуально при решении задачи сканирования помещения. А также этот алгоритм наиболее подходит для реконструкции сложных объектов.

3.2. Выполнена программная реализация реконструкции сцены помещения по одному изображению (восстановления поверхности). Для данной задачи был рассмотрен новый способ, не используемый ранее в данной работе, вычисления трехмерного облака точек с помощью предсказанной карты глубины.

3.3. Проведен ряд экспериментов и проанализированы результаты запуска программной реализации на фотографиях реальных помещений. Для каждого из примера была получена карта глубины, трёхмерное облако точек, граничное облако точек (граничное облако точек было построено на основе карты глубины, у которой ненулевые значения лежат только на границе). И в конечном итоге, на основе трехмерного облака точек была восстановлена трехмерная модель сцены.

Список литературы

1. Дорохов Д.С. Овчинников Н.В. Взаимодействие технологий информационного моделирования с возможностями виртуальной и дополненной реальности // Вестник евразийской науки. – 2022. – № 3. – Т. 14.
2. Брызгалина В.В. Перспективы использования технологий дополненной реальности в сфере строительства и проектирования зданий / В.В. Брызгалина, Д.А. Протопопова // Актуальные проблемы науки и техники. – 2022. – С. 201-202.
3. Ельчищева Т.Ф. Черных А.В. 3D-моделирование на основе технологии LIDAR / Сборник докладов VII Международной научно-практической конференции, посвященной 170-летию В.Г. Шухова. Белгород, 2023. – С. 130 – 136.
4. Попова Е. И. Роль 3D-сканирования в расследовании преступлений / Е. И. Попова // Научное обеспечение раскрытия, расследования и предупреждения преступлений. – 2023. – С. 150-155.
5. Безменов В. М. Фотограмметрия: учебное пособие / В. М. Безменов – Москва: Директ-Медиа, 2023. – С. 10 – 198. – ISBN 978-5-4499-3584-7.
6. Коньков В.В. Программно-аппаратный комплекс получения фотоизображений на основе технологии IIOT и анализ точности различных алгоритмов цифровой генерации 3D моделей на основе принципа фотограмметрии / В.В. Коньков, А.Б. Замчалов, М.Г. Жабицкий // INTERNATIONAL JOURNAL OF OPEN INFORMATION TECHNOLOGIES – 2023. – №8. – Т. 11. – С. 32 – 48.
7. Тупиков В.А. Программное обеспечение для создания трехмерных сцен тестирования алгоритмов компьютерного зрения / В.А. Тупиков, В.А. Павлова, А.И. Лизин, А.В. Колдомасов, П.А. Гессен // Известия ТулГУ. – 2023. – № 6. – С. 51 – 65.
8. Гадасин Д.В. Трехмерная реконструкции объекта по одному изображению с использованием глубоких свёрточных нейронных сетей / Шведов А.В., Кузин И.А. // Т-COMM: телекоммуникации и транспорт. – 2022. - №7. – т. 16. – с. 29-35.
9. Даминов, И. А. Комбинирование алгоритмов SfM и ORB при 3D-реконструкции / И. А. Даминов, А. Ю. Арсенюк, А. С. Тощев // Электронные библиотеки. – 2023. – Т. 26, № 4. – С. 456-465.

10. Лукьянец П.Е. Исследование современных систем распознавания объектов: методы обнаружения, этапы развития, наиболее популярные детекторы / П.Е. Лукьянец, П.Ю. Парфишов, Л.А. Решетов, М.В. Соколовская // Моделирование и ситуационное управление качеством сложных систем – 2023. – С. 49 – 55.
11. Kazhdan M. Poisson Surface Reconstruction / Kazhdan M., Bolitho M., Hoppe H. // Symposium on Geometry Processing - 2006.

Приложение А. PlaneTracking

```
namespace App.Code
{
    public class ARPlaceObject: MonoBehaviour
    {
        public ARRaycastManager RayCastManager;

        public GameObject CubeToPlace;
        public GameObject SphereToPlace;
        private List<ARRaycastHit> _hits;
        private List<GameObject> _instances;

        public void Awake()
        {
            _instances = new List<GameObject>();

            _hits = new List<ARRaycastHit>();
        }

        private void SpawnPrefab(GameObject obj)
        {
            _instances.Add(Instantiate(obj, _hits[0].pose.position,
                  _hits[0].pose.rotation));
        }

        private void Update()
        {
            if (Input.touchCount == 0) return;

            if (Input.GetTouch(0).phase != TouchPhase.Ended) return;

            var screenPosition = Input.GetTouch(0).position;
            var result = RayCastManager.Raycast(screenPosition, _hits,
                   TrackableType.PlaneWithinBounds);

            if (result == true && _hits[0].trackable is ARPlane plane)
            {
                if (plane.alignment == PlaneAlignment.Vertical)
                {
                    SpawnPrefab(CubeToPlace);
                }
                else
                {
                    SpawnPrefab(SphereToPlace);
                }
            }
        }
    }
}
```

Приложение B. Метод stereoRectifyUncalibrated + StereoBM и метод плотного потока

```
import numpy as np
from matplotlib import pyplot as plt

import cv2

image1_jpg = cv2.imread(r'E:\BKP(mag)\Habr\1.jpg', 0)
image2_jpg = cv2.imread(r'E:\BKP(mag)\Habr\2.jpg', 0)

def plot(title_str, image, index):
    plt.subplot(2, 2, index)
    plt.title(title_str)
    plt.imshow(image, 'gray')
    plt.gca().get_xaxis().set_visible(False)
    plt.gca().get_yaxis().set_visible(False)

def Method_StereoDepthMap(image1_jpg, image2_jpg)
    orb_object = cv2.ORB_create()
    keypointsList_img1, descriptors_img1 = orb_object.detectAndCompute(image1_jpg, None)
    keypointsList_img2, descriptors_img2 = orb_object.detectAndCompute(image2_jpg, None)
    if len(keypointsList_img1) == 0 or len(keypointsList_img2) == 0 or descriptors_img1 is None or
descriptors_img2 is None:
        print("No keypoints or descriptors found.")
        return None

    matcher_bf_object = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

    matches = matcher_bf_object.match(descriptors_img1, descriptors_img2)
    matches = sorted(matches, key=lambda x: x.distance)

    source_points = np.vstack([np.array(keypointsList_img1[m.queryIdx].pt) for m in matches])
    destination_points = np.vstack([np.array(keypointsList_img2[m.trainIdx].pt) for m in matches])

    Fundamental_matrix, mask = cv2.findFundamentalMat(source_points, destination_points)

    _, left_rectification_matrix , right_rectification_matrix =
cv2.stereoRectifyUncalibrated(source_points.reshape(source_points.shape[ 0], 1, 2),
destination_points.reshape(destination_points.shape[0], 1, 2), Fundamental_matrix, image1_jpg.shape)
    if left_rectification_matrix  is None or right_rectification_matrix is None:
        print("Stereo rectification failed.")
        return None

    rect1 = cv2.warpPerspective(image1_jpg, left_rectification_matrix , (852, 480))
    rect2 = cv2.warpPerspective(image2_jpg, right_rectification_matrix, (852, 480))

    stereo = cv2.StereoBM_create(numDisparities=16, blockSize=15)
    return stereo.compute(rect1, rect2)

def Method_OpticalFlowDepthMap(image1_jpg, image2_jpg):
    vec_optical_flow = cv2.calcOpticalFlowFarneback(image1_jpg, image2_jpg, None, .05, 1, 12, 2, 8,
1.2, 0)
    module_vector_speed,    angular_vec_speed    =    cv2.cartToPolar(vec_optical_flow[...,    0],
vec_optical_flow[..., 1])
    return module_vector_speed
```

Приложение С. Добавление стен, пола и потолка

```
using System.Collections.Generic;
using UnityEngine;
using System.IO;

namespace App.Code
{
    struct Wall
    {
        float position_x;
        float position_y;
        float position_z;
        public float scale_x;
        public float scale_y;
        public float scale_z;

        public float distance;

        public Wall(float posX, float posY, float posZ, float scaleX, float scaleY, float scaleZ,
            float _distance)
        {

            this.position_x = posX;
            this.position_y = posY;
            this.position_z = posZ;

            scale_x = scaleX;
            scale_y = scaleY;
            scale_z = scaleZ;

            distance = _distance;
        }
    }

    public class AddObjects : MonoBehaviour
    {
        private static List<GameObject> objects_list;
        private static List<Wall> walls;
        public GameObject wall_gameObj;

        private static float globalScale = 5f;


        public void Awake()
        {
            objects_list = new List<GameObject>();
            walls = new List<Wall>();
        }

        Wall getWall(string[] depiction)
        {
            float tmp_posX = float.Parse((depiction[1])) * globalScale;
            float tmp_posY = float.Parse((depiction[2])) * globalScale;
            float tmp_posZ = float.Parse((depiction[3])) * globalScale;

            float tmp_scaleX = float.Parse((depiction[5])) * globalScale;
            float tmp_scaleY = float.Parse((depiction[6])) * globalScale;
            float tmp_scaleZ = float.Parse((depiction[7])) * globalScale;

            float tmp_distance = float.Parse((depiction[8])) * globalScale;

            Wall tmp_wall = new Wall(tmp_posX, tmp_posY, tmp_posZ, tmp_scaleX,
                tmp_scaleY, tmp_scaleZ, tmp_distance);

            return tmp_wall;
        }

        public void AddObject(GameObject obj, Vector3 pos, Quaternion quat,
```

```
            Vector3 scale)
        {
            objects_list.Add(Instantiate(obj, pos, quat));
            Vector3 newScale_wall = scale;
            objects_list[objects_list.Count - 1].transform.localScale =
                newScale_wall;
        }

        public void AddWall()
        {
            string folderPath = "//Assets//Room.txt";
            string[] lines = File.ReadAllLines(folderPath);

            foreach (string line in lines)
            {
                string[] values = line.Split(';');

                if (values[0] == "wall")
                {
                    Wall tmp_wall = getWall(values);
                    walls.Add(tmp_wall);
                }
            }

            AddObject(wall_gameObj, new Vector3(0f, 2.5f, 0f + walls[0].distance), new Quaternion(0,
                  0, 0, 1), new Vector3(walls[0].scale_x, walls[0].scale_y, walls[0].scale_z));
            AddObject(wall_gameObj, new Vector3(0f, 2.5f, 0f - walls[0].distance), new Quaternion(0,
                  0, 0, 1), new Vector3(walls[0].scale_x, walls[0].scale_y, walls[0].scale_z));
            AddObject(wall_gameObj, new Vector3(walls[0].scale_x / 2, 2.5f, 0f), new Quaternion(0, -
                  0.70f, 0f, 0.70f), new Vector3(walls[2].scale_x, walls[2].scale_y,
                  walls[2].scale_z));
            AddObject(wall_gameObj, new Vector3(-walls[0].scale_x / 2, 2.5f, 0f), new Quaternion(0,
                  0.70f, 0, 0.70f), new Vector3(walls[2].scale_x, walls[2].scale_y,
                  walls[2].scale_z));
        }
    }
}
```

Приложение D. Визуализация комнаты (по точкам)

```
def Method_OpticalFlowDepthMap(image1_jpg, image2_jpg):
    vec_optical_flow = cv2.calcOpticalFlowFarneback(image1_jpg, image2_jpg, None, 0.5, 3, 20, 10, 5,
1.2, 0)
    module_vector_speed, angular_vec_speed = cv2.cartToPolar(vec_optical_flow[..., 0],
vec_optical_flow[..., 1])
    return vec_optical_flow, module_vector_speed, angular_vec_speed

def plot(title_str, image, index):
    plt.subplot(2, 2, index)
    plt.title(title_str)
    plt.imshow(image, 'gray')
    plt.gca().get_xaxis().set_visible(False)
    plt.gca().get_yaxis().set_visible(False)

def convert_ply(points):
    with open(r'F:\Diplom\BKP(mag)\Habr\points.ply', 'w') as f:
        f.write('ply\n')
        f.write('format ascii 1.0\n')
        f.write('element vertex {}\n'.format(len(points)))
        f.write('property float x\n')
        f.write('property float y\n')
        f.write('property float z\n')
        f.write('end_header\n')
        for point in points:
            f.write('{:.2f} {:.2f} {:.2f}\n'.format(point[0], point[2], point[1]))

B = 1
B_step = 1
w_width = 1920
h_heigth = 1080
f_focalLength = 250

Z = 31
delta = 1.4

image1_jpg = cv2.imread(r'F:\Diplom\BKP(mag)\Habr\1.jpg', 0)
image2_jpg = cv2.imread(r'F:\Diplom\BKP(mag)\Habr\2.jpg', 0)

def get_point_cloud_2D():
    vec_optical_flow = cv2.calcOpticalFlowFarneback(image1_jpg, image2_jpg, None, 0.5, 3, 20, 10, 5,
1.2, 0)
    module_vector_speed, angular_vec_speed = cv2.cartToPolar(vec_optical_flow[..., 0],
vec_optical_flow[..., 1])

    conture = cv2.Canny(image1_jpg, 100, 200)

    point_cloud2D = []
    for y in range(image1_jpg.shape[0]):
        for x in range(image1_jpg.shape[1]):
            if conture[y, x] == 0:
                continue
            #delta = module_vector_speed[y, x]
            #if delta == 0:
                #continue
            Z_axes = (B * f_focalLength) / delta
            X_axes = (Z * (x - h_heigth / 2.)) / f_focalLength
            Y_axes = (Z * (y - h_heigth / 2.)) / f_focalLength
            tmp_point = np.array([X_axes, Y_axes, Z_axes])
            point_cloud2D.append(tmp_point)

    return point_cloud2D
```

```
def get_point_cloud_3D():
    vec_optical_flow = cv2.calcOpticalFlowFarneback(image1_jpg, image2_jpg, None, 0.5, 3, 20, 10, 5,
1.2, 0)
    module_vector_speed, angular_vec_speed = cv2.cartToPolar(vec_optical_flow[..., 0],
vec_optical_flow[..., 1])

    conture = cv2.Canny(image1_jpg, 100, 200)

    point_cloud3D = []

    for y in range(image1_jpg.shape[0]):
        for x in range(image1_jpg.shape[1]):
            if conture[y, x] == 0:
                continue
            delta = angular_vec_speed[y, x]
            if delta == 0:
                continue
            Z_axes = (B * f_focalLength) / delta
            X_axes = (Z * (x - h_heigth / 2.)) / f_focalLength
            Y_axes = (Z * (y - h_heigth / 2.)) / f_focalLength
            tmp_point = np.array([X_axes, Y_axes, Z_axes])
            point_cloud3D.append(tmp_point)

    for y in range(image1_jpg.shape[0]):
        for x in range(image1_jpg.shape[1]):
            if conture[y, x] == 0:
                continue
            delta = module_vector_speed[y, x]
            if delta == 0:
                continue
            Z_axes = (B * f_focalLength) / delta
            X_axes = (Z * (x - h_heigth / 2.)) / f_focalLength
            Y_axes = (Z * (y - h_heigth / 2.)) / f_focalLength
            tmp_point = np.array([X_axes, Y_axes, Z_axes])
            point_cloud3D.append(tmp_point)

    for y in range(image1_jpg.shape[0]):
        for x in range(image1_jpg.shape[1]):
            if conture[y, x] == 0:
                continue
            #delta = module_vector_speed[y, x]
            #if delta == 0:
                #continue
            Z_axes = (B * f_focalLength) / delta
            X_axes = (Z * (x - h_heigth / 2.)) / f_focalLength
            Y_axes = (Z * (y - h_heigth / 2.)) / f_focalLength
            tmp_point = np.array([X_axes, Y_axes, Z_axes])
            point_cloud3D.append(tmp_point)

    point_cloud3D = np.vstack(point_cloud3D)
    return point_cloud3D

conture1 = cv2.Canny(image1_jpg, 100, 200)
conture2 = cv2.Canny(image2_jpg, 100, 200)

plot('img_left', conture1, 1)
plot('img_right', conture2, 2)

points_2d = get_point_cloud_2D()
points_3D = get_point_cloud_3D()

convert_ply(points_2d)
#convert_ply(points_3D)

filtered_points3D = []
for i in range(len(points_3D)):
    if(23 <= points_3D[i][2] <= 91):
        filtered_points3D.append(points_3D[i])

convert_ply(filtered_points3D)
```

Приложение Е. Алгоритм кластеризации DBSCAN

```
def distance2D(point1, point2):
        return math.sqrt((point2[0] - point1[0])**2 + (point2[1] - point1[1])**2)

def find_neighbours(points_set, point1, eps):
    neighbours = []
    for point2 in points_set:
        if distance2D(point1, point2) < eps:
            neighbours.append(point2)
    return neighbours

def update_cluster(p, neighbours, clusters, clustered_points, visited_points, eps, m, points_set,
NOISE, C):
    if C not in clusters:
        clusters[C] = []
    clusters[C].append(p)
    clustered_points.add(p)
    while neighbours:
        q = neighbours.pop()
        if q not in visited_points:
            visited_points.add(q)
            neighbourz = find_neighbours(points_set, q, eps)
            if len(neighbourz) > m:
                neighbours.extend(neighbourz)
        if q not in clustered_points:
            clustered_points.add(q)
            clusters[C].append(q)
            if q in clusters[NOISE]:
                clusters[NOISE].remove(q)

def dbscan_naive(points_set, eps, m, distance2D):
    NOISE = 0
    C = 1
    visited_points = set()
    clustered_points = set()
    clusters = {NOISE: []}

    for point1 in points_set:
        if point1 in visited_points:
            continue
        visited_points.add(point1)
        neighbours = find_neighbours(points_set, point1, eps)
        if len(neighbours) < m:
            clusters[NOISE].append(point1)
        else:
            C += 1
            update_cluster(point1, neighbours, clusters, clustered_points, visited_points, eps, m,
points_set, NOISE, C)

    return clusters

if __name__ == "__main__":
    P = [(random.randn()/2, random.randn()/6) for i in range(150)]
    P.extend([(random.randn()/3 + 2.5, random.randn()/5) for i in range(150)])
    P.extend([(i/25 - 1, + random.randn()/20 - 1) for i in range(100)])
    clusters = dbscan_naive(P, 0.2, 4, lambda x, y: hypot(x[0] - y[0], x[1] - y[1]))
    for c, points in zip(cycle('bgrcmykgrcmykgrcmykgrcmykgrcmykgrcmyk'), clusters.values()):
        X = [p[0] for p in points]
        Y = [p[1] for p in points]
        plt.scatter(X, Y, c=c)
    plt.show()
```

```
def find_eps():
    random_eps_arr = np.random.randint(0, 101, size=1000)
    for epsilon in random_eps_arr:
        dbscan = DBSCAN(eps = epsilon, min_samples = 20)
        dbscan.fit(points_2d)
        labels = dbscan.labels_
        unique_labels = np.unique(labels)
        if(len(unique_labels) == 4):
            return epsilon

dbscan = DBSCAN(eps = 8, min_samples = 20)
dbscan.fit(points_2d)

labels = dbscan.labels_

epsilon = find_eps()
epsilon

dbscan = DBSCAN(eps = epsilon, min_samples = 20)
dbscan.fit(points_2d)
labels = dbscan.labels_
unique_labels = np.unique(labels)

clusters = [[] for _ in range(0, len(unique_labels + 1))]

for i in range(len(labels)):
    label = labels[i]
    clusters[label - 1].append(points_2d[i])

for i in range(len(clusters)):
    clusters[i] = np.array(clusters[i])

len(clusters)ts3D[i]]])
```

Приложение F. Алгоритм классификации на основе геометрических характеристик

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Distance:
    def __init__(self):
        self.distance = 0.0
        self.angle = 0.0
        self.index = 0
        self.position = 0.0

points_normalized

two_dim_array = points_normalized[:, :2]

# Преобразование массива в формат source_points
source_points = [Point(x, y) for x, y in two_dim_array]

# Расчет центра фигуры
center_x = sum([point.x for point in source_points]) / len(source_points)
center_y = sum([point.y for point in source_points]) / len(source_points)

# Создаем фигуру и оси для графика
fig, ax = plt.subplots()

# Рисуем точки
for point in source_points:
  circle = Circle((point.x, point.y), radius=0.001, color='blue') # Создаем круг для точки
  ax.add_patch(circle) # Добавляем круг на график

# Рисуем центр
ax.plot(center_x, center_y, 'ro') # Рисуем красный кружок в центре

# Отображаем график
plt.show()

center_x, center_y

# Расчет расстояний от точек до центра
distancies = []
for index, point in enumerate(source_points):
    distance = Distance()
    delta_x = point.x - center_x
    delta_y = point.y - center_y
    distance.distance = delta_x ** 2 + delta_y ** 2
    distance.angle = np.arctan2(delta_x, delta_y)
    distance.index = index
    distancies.append(distance)

# Массив расстояний
distances_to_center = [d.distance for d in distancies]
indexes = [i.index for i in distancies]

# Построение графика
plt.plot(indexes, distances_to_center, marker='o', markersize=1, label='Расстояние до центра')  #
Уменьшите markersize
plt.title('Расстояние от каждой точки до центра')
plt.xlabel('Индекс точки')
plt.ylabel('Расстояние до центра')
plt.grid(True)
```

```
# Добавление линии для максимального расстояния
max_distance = max(distances_to_center)
plt.axhline(y=max_distance, color='r', linestyle='--', linewidth=1, label=f'Max:
{max_distance:.2f}')

# Добавление легенды
plt.legend()

plt.show()

#min(distances_to_center)

count = 0
for i in range(len(distances_to_center)):
    if abs(distances_to_center[i] - max(distances_to_center)) < 0.01:
        count += 1
        #print(source_points[i].x, source_points[i].y)

distancies.sort(key=lambda d: d.angle)

# Нормализация расстояний, чтобы дальнейшие расчеты не зависели от размера фигуры.
minimal_angle = float('inf')
maximal_angle = float('-inf')
maximal_distance = float('-inf')

for distance in distancies:
    minimal_angle = min(distance.angle, minimal_angle)
    maximal_angle = max(distance.angle, maximal_angle)
    maximal_distance = max(distance.distance, maximal_distance)

for distance in distancies:
    distance.distance /= maximal_distance
    distance.position = len(source_points) * (distance.angle - minimal_angle) / (maximal_angle -
minimal_angle)

# Определение функции бинарного поиска
def binary_search(distancies, target):
    low, high = 0, len(distancies) - 1
    while low <= high:
        mid = (low + high) // 2
        if distancies[mid].position < target.position:
            low = mid + 1
        elif distancies[mid].position > target.position:
            high = mid - 1
        else:
            return mid
    return -low - 1  # Возвращаем отрицательное значение, если не найдено

# Вычисление массива расстояний, в котором угол будет изменяться линейно относительно индекса в
массиве.
for index in range(len(source_points)):
    dd = Distance()
    dd.position = index

    # Бинарный поиск
    found = binary_search(distancies, dd)

    if found >= 0:
        source_points[index] = distancies[found].distance
    else:
        point = -found - 1
        left = distancies[point - 1]
        right = distancies[point]
        factor = (index - left.position) / (left.position - right.position)

        source_points[index] = left.distance + factor * (left.distance - right.distance)

        if factor < 0.5:
            indexes[index] = point - 1
        else:
            indexes[index] = point
```

```
#Получаем массивы углов и расстояний
angles = [d.angle for d in distancies]
distances_to_center = [d.distance for d in distancies]

# Построение графика зависимости расстояния от угла
plt.figure(figsize=(8, 6))
plt.plot(angles, distances_to_center, marker='o', linestyle='-', color='b', linewidth=0.1) #
Уменьшаем толщину линии
plt.title('Зависимость расстояния до центра от угла (радианы)')
plt.xlabel('Угол (радианы)')
plt.ylabel('Расстояние до центра (в пикселях)')
plt.grid(True)
plt.show()

# Функция для обнаружения пиков
def peak_detect(data_size):
    size = data_size // (2 * 5 + 1)
    center = (size // 2) | 1
    values = np.zeros(size)
    kernel_sum = 0.0

    for index in range(size):
        factor = abs(index - center) / (center + 1.0)
        values[index] = np.sqrt((1 - factor) ** 2 + 1) - 1
        kernel_sum += values[index]

    values -= kernel_sum / size

    return Kernel(values, center)

class Kernel:
    def __init__(self, values, center):
        self.values = values
        self.center = center

def create_kernel(size):
    kernel_values = np.zeros(size)
    kernel_sum = 0.0
    center = size // 2

    for index in range(size):
        factor = abs(index - center) / (center + 1.0)
        kernel_values[index] = np.sqrt((1 - factor) ** 2 + 1) - 1
        kernel_sum += kernel_values[index]

    # Нормируем ядро
    kernel_values += -kernel_sum / size
    return Kernel(kernel_values, center)

# Применение свертки
def apply_kernel(data, kernel):
    result = np.zeros(len(data))
    for data_index in range(len(data)):
        sum_value = 0.0
        for kernel_index in range(len(kernel.values)):
            offset = data_index + kernel_index - kernel.center
            if offset < 0:
                offset += len(data)
            if offset >= len(data):
                offset -= len(data)
            sum_value += kernel.values[kernel_index] * data[offset]
        result[data_index] = sum_value
    return result

# Создаем ядро для свертки
kernel_size = 11  # Размер ядра
kernel = create_kernel(kernel_size)

# Применяем свертку к нормализованным расстояниям
convolved_values = apply_kernel([d.distance for d in distancies], kernel)
```

```
# Масштабируем углы для графика
scaled_angles = np.linspace(0, 1024, num=len(convolved_values))

# Построение графика
plt.figure(figsize=(10, 6))
plt.plot(scaled_angles, convolved_values, marker='o', linestyle='-', color='b')
plt.title('Зависимость свертки расстояния до центра от угла (отмасштабировано до 1024)')
plt.xlabel('Угол (отмасштабированный до 1024)')
plt.ylabel('Значение свертки')
plt.grid(True)
plt.show()
```

Приложение G. Обработка облака точек для классификации

Получение массива точек:

```
# Преобразование массива в формат source_points
two_dim_array = points_normalized[:, :2]
source_points = [Point(x, y) for x, y in two_dim_array]

len(source_points)
```

Удалим процент точек:

```
# Доля точек, которые нужно оставить (40%)
fraction_to_keep = 0.05

# Вычисляем количество точек для сохранения
num_points_to_keep = int(len(source_points) * fraction_to_keep)

# Выбираем случайные точки, чтобы сохранить их
source_points = random.sample(source_points, num_points_to_keep)
```

Посмотрим на изменения:

```
len(source_points)

# Расчет центра фигуры
center_x = sum([point.x for point in source_points]) / len(source_points)
center_y = sum([point.y for point in source_points]) / len(source_points)
center_x, center_y

# Создаем фигуру и оси для графика
fig, ax = plt.subplots()

# Рисуем точки
for point in source_points:
  circle = Circle((point.x, point.y), radius = 0.001, color='blue') # Создаем круг для точки
  ax.add_patch(circle) # Добавляем круг на график

# Рисуем центр
ax.plot(center_x, center_y, 'ro') # Рисуем красный кружок в центре

# Отображаем график
plt.show()
```

Объединим общие точки:

```
def distance(point1, point2):
    return math.sqrt((point1.x - point2.x)**2 + (point1.y - point2.y)**2)

# Значение epsilon
eps = 1e-6

# Создаем новый список для точек, которые пройдут проверку
filtered_points = []

for i, point in enumerate(source_points):
    # Проверяем расстояние до каждой другой точки
    has_close_neighbor = False
    for j, other_point in enumerate(source_points):
        if i != j and distance(point, other_point) < eps:
            has_close_neighbor = True
            break

    # Если близкого соседа нет, добавляем точку в отфильтрованный список
    if not has_close_neighbor:
        filtered_points.append(point)

# Перезаписываем source_points, чтобы оставить только отфильтрованные точки
```

```
source_points = filtered_points
```

Посмотрим на изменения:

```
# Расчет центра фигуры
center_x = sum([point.x for point in source_points]) / len(source_points)
center_y = sum([point.y for point in source_points]) / len(source_points)
center_x, center_y

# Создаем фигуру и оси для графика
fig, ax = plt.subplots()

# Рисуем точки
for point in source_points:
  circle = Circle((point.x, point.y), radius=0.001, color='blue') # Создаем круг для точки
  ax.add_patch(circle) # Добавляем круг на график

# Рисуем центр
ax.plot(center_x, center_y, 'ro') # Рисуем красный кружок в центре

# Отображаем график
plt.show()

len(source_points)
```

Удаление точек на основании среднего расстояния до центра:

```
# Расчет расстояний от точек до центра
distancies = []
for index, point in enumerate(source_points):
    distance = Distance()
    delta_x = point.x - center_x
    delta_y = point.y - center_y
    distance.distance = delta_x ** 2 + delta_y ** 2
    distance.angle = np.arctan2(delta_x, delta_y)
    distance.index = index
    distancies.append(distance)

# Массив расстояний
distances_to_center = [d.distance for d in distancies]
indexes = [i.index for i in distancies]

# Вычисление среднего расстояния до центра
mean_distance = np.mean(distances_to_center)
print (mean_distance)

# Задайте значение eps
eps = 0.2

# Удаление точек, у которых расстояние отличается от среднего более чем на eps
source_points = [
    source_points[dist.index] for dist in distancies
    if abs(dist.distance - mean_distance) <= eps
]
```

Посмотрим на изменения:

```
# Количество оставшихся точек
len(source_points)

# Расчет центра фигуры
center_x = sum([point.x for point in source_points]) / len(source_points)
center_y = sum([point.y for point in source_points]) / len(source_points)
center_x, center_y

# Создаем фигуру и оси для графика
fig, ax = plt.subplots()

# Рисуем точки
for point in source_points:
  circle = Circle((point.x, point.y), radius=0.001, color='blue') # Создаем круг для точки
```

```
  ax.add_patch(circle) # Добавляем круг на график

# Рисуем центр
ax.plot(center_x, center_y, 'ro') # Рисуем красный кружок в центре

# Отображаем график
plt.show()
```

Приложение Н. Реконструкция помещения

```
import torch
from transformers import GLPNImageProcessor, GLPNForDepthEstimation

import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import open3d as o3d

def load_and_prepare_image(image_path):
    image = Image.open(image_path)
    h, w = 0, 0

    if image.height > 480:
        h = 480
    else:
        h = image.height

    h = h - (h % 32)
    w = int(h * image.width / image.height)

    diff = w % 32
    if diff < 16:
        w = w - diff
    else:
        w + 32 - diff

    return image.resize((w, h))

def predict_depth(model, inputs):
    with torch.no_grad():
        outputs = model(**inputs)
        return outputs.predicted_depth

def remove_borders(depth_map, image, pad = 16):
    depth_map = depth_map.squeeze().cpu().numpy() * 1000.0
    depth_map = depth_map[pad:-pad, pad:-pad]
    cropped_image = image.crop((pad, pad, image.width - pad, image.height - pad))

    return depth_map, cropped_image

def visualize_results(image, depth_map):
    fig, ax = plt.subplots(1, 2)

    ax[0].set_title("Original Image")
    ax[1].set_title("Depth Map")

    ax[0].imshow(image)
    ax[0].tick_params(left = False, bottom = False, labelleft = False, labelbottom = False)
    ax[1].imshow(depth_map, cmap = 'jet')
    ax[1].tick_params(left = False, bottom = False, labelleft = False, labelbottom = False)
    plt.tight_layout()
    plt.pause(5)

def create_point_cloud(image, depth_map):
    depth_image = (depth_map * 255 / np.max(depth_map)).astype('uint8')
    image_array = np.array(image)

    depth_o3d = o3d.geometry.Image(depth_image)
    image_o3d = o3d.geometry.Image(image_array)
    image_RGBD = o3d.geometry.RGBDImage.create_from_color_and_depth(image_o3d, depth_o3d,
convert_rgb_to_intensity = False)

    camera_intrinsic = o3d.camera.PinholeCameraIntrinsic()
    camera_intrinsic.set_intrinsics(image.width, image.height, 500, 500, image.width * 0.5,
image.height * 0.5)

    point_cloud = o3d.geometry.PointCloud.create_from_rgbd_image(image_RGBD, camera_intrinsic)
```

```
    cleaned_point_cloud, indexes = point_cloud.remove_statistical_outlier(nb_neighbors = 20,
std_ratio = 20.0)
    point_cloud = point_cloud.select_by_index(indexes)

    print(f"Количество точек: {len(point_cloud.points)}")
    print(f"Количество точек после удаления выбросов: {len(cleaned_point_cloud.points)}")
    #print(f"Индексы оставшихся точек: {indexes}")

    point_cloud.estimate_normals()
    point_cloud.orient_normals_to_align_with_direction()

    return point_cloud

def reconstruct_surface(point_cloud):
    mesh = o3d.geometry.TriangleMesh.create_from_point_cloud_poisson(point_cloud, depth = 10,
n_threads = 8)[0]
    rotation = mesh.get_rotation_matrix_from_xyz((3.14159, 0, 0))
    mesh.rotate(rotation, center = (0, 0, 0))
    return mesh

def save_and_visualize_mesh(mesh, point_cloud, output_path):
    o3d.io.write_triangle_mesh(output_path, mesh)
    o3d.visualization.draw_geometries([point_cloud])

def main():
    image = load_and_prepare_image("F://00549_colors.png")

    #  Инициализация модели для оценки глубины и экстрактора для подготавки изображения
    feature_extractor = GLPNImageProcessor.from_pretrained("vinvino02/glpn-nyu")
    model = GLPNForDepthEstimation.from_pretrained("vinvino02/glpn-nyu")
    x_inputs =  feature_extractor(images = image, return_tensors="pt")
    predicted_depth = predict_depth(model, x_inputs)

    depth_map, cropped_image = remove_borders(predicted_depth, image)
    visualize_results(cropped_image, depth_map)

    point_cloud = create_point_cloud(cropped_image, depth_map)
    mesh = reconstruct_surface(point_cloud)
    save_and_visualize_mesh(mesh, point_cloud, 'D://mesh1.obj')
```