МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра веб-технологий и компьютерного моделирования

НИКОЛЬСКИЙ

Никита Юрьевич

АЛГОРИТМЫ КОМПЬЮТЕРНОГО ЗРЕНИЯ И РАСПОЗНАВАНИЕ ШТРИХ-КОДОВ НА РАСТРОВЫХ ИЗОБРАЖЕНИЯХ

Дипломная работа

Научный руководитель:
кандидат физ.-мат. наук,
доцент Б.М. Дубров

Допущен к защите

«___» _____ 2017 г.

Зав. кафедрой веб-технологий и компьютерного моделирования

кандидат физ.-мат. наук, доцент В.С. Романчик

Минск, 2017

ОГЛАВЛЕНИЕ

РЕФЕРАТ

Дипломная работа содержит в пояснительной записке 53 страницы, 44 рисунка, 8 источников литературы.

КОМПЬЮТЕРНОЕ ЗРЕНИЕ, АЛГОРИТМЫ КОМПЬЮТЕРНОГО ЗРЕНИЯ, ШТРИХ-КОДЫ, РАСПОЗНАВАНИЕ ШТРИХ-КОДОВ НА РАСТРОВЫХ ИЗОБРАЖЕНИЯ

Целью данной дипломной работы является написание приложения для распознавания штрих-кодов на растровых изображениях. Во введении проводится анализ актуальности темы данной дипломной работы, выявляются его цели и задачи, рассматриваются объекты и предметы исследования, определяется, в чем заключается научная новизна данного исследования, теоретическая значимость и практическая ценность. В первой главе даётся общее понятие компьютерного зрения и алгоритмов для работы с ним, а также вводится определение штрих-кодов и их использовании в современном мире. Во второй главе описываются используемые технологии и архитектура приложения, рассматривается функционал работы библиотеки для распознавания штрих-кодов QuaggaJS, анализируется работа алгоритма, который она реализует. Далее приводится пример реализации алгоритма с использованием языка программирования python и библиотеки OpenCV.

THE ABSTRACT

Thesis contains in the explanatory note 53 pages, 44 drawings, 8 sources of literature.

COMPUTER VISION, COMPUTER VISION ALGORITHMS, BARCODES, BARCODE RECOGNITION ON RASTER IMAGES

The purpose of this thesis is writing an application for recognizing barcodes on raster images. The introduction analyzes the relevance of the topic of this thesis work, identifies its goals and objectives, examines the objects and subjects of research, determines what is the scientific novelty of this study, its theoretical significance and practical value. The first chapter gives a general concept of computer vision and algorithms for working with it, and also introduces the definition of bar codes and their use in the modern world. The second chapter describes the technologies used and the application architecture, discusses the library's functionality for recognizing the QuaggaJS barcodes, analyzes the work of the algorithm that it implements. The following is an example of implementing an algorithm using the python programming language and the OpenCV library.

РЭФЕРАТ

Дыпломная праца ўтрымлівае ў тлумачальнай запісцы 53 старонкі, 44 малюнка, 8 крыніц літаратуры, прыкладанне.

КАМПУТАРНАЕ ГЛЕДЖАННЕ, АЛГАРЫТМЫ КАМПУТАРНАГА ГЛЕДЖАННЯ, ШТРЫХ-КОДЫ, РАСПАЗНАВАННЕ ШТРЫХ-КОДАЎ НА РАСТРАВЫХ МАЛЮНКАХ

Мэтай дадзенай дыпломнай працы з'яўляецца напісанне прыкладання для распазнання штрых-кодаў на растравых малюнках. Ва ўвядзенні праводзіцца аналіз актуальнасці тэмы дадзенай дыпломнай працы, выяўляюцца яго мэты і задачы, разглядаюцца аб'екты і прадметы даследавання, вызначаецца, у чым заключаецца навуковая навізна дадзенага даследавання, тэарэтычная значнасць і практычная каштоўнасць. У першай чале даецца агульнае паняцце камп'ютэрнага гледжання і алгарытмаў для працы з ім, а таксама ўводзіцца вызначэнне штрых-кодаў і іх выкарыстанні ў сучасным свеце. У другой чале апісваюцца выкарыстаныя тэхналогіі і архітэктура прыкладання, разглядаецца функцыянал працы бібліятэкі для распазнання штрых-кодаў QuaggaJS, аналізуецца праца алгарытму, які яна рэалізуе. Далей прыводзіцца прыклад рэалізацыі алгарытму з выкарыстаннем мовы праграмавання python і бібліятэкі OpenCV.

СПИСОК ИСПОЛЬЗУЕМЫХ ОБОЗНАЧЕНИЙ И ТЕРМИНОВ

МО - машинное обучение

КЗ - компьютерное зрение

АКЗ - алгоритмы компьютерного зрения

JS – JavaScript

npm – node package manager

ВВЕДЕНИЕ

Зрение – является важным источником информации для человека. Благодаря ему мы получаем по разным оценкам от 70% до 90% знаний об окружающем нас мире. Человек воспринимаем трехмерную структуру окружающего мира легко и даже не задумывается, какие сложные процессы происходят в его голове. Сидя на столе, мы можем описать цветок, который стоит перед нами, можем указать форму и полупрозрачность каждого лепестка. Глядя на групповой портрет в рамке, мы также можем подсчитать и даже назвать всех людей, которые изображены на нём, описать их эмоции и внешность.

Компьютерное зрение – это один из самых сложных процессов, который человек пытался когда-либо понять, а от сюда следует, что и разработать автоматизированные системы использующие его возможности крайне трудно.

Существует множество алгоритмов, благодаря которым компьютер может распознавать объекты на изображении с той или иной степенью точности. Для каждого из алгоритмов есть какие-то границы его корректной работы. Исследователи в области компьютерного зрения разрабатывают математические методы, которые помогут воссоздать трёхмерную модель объекта, основываясь на его двухмерном изображении. Так же современные наработки позволяют компьютеру распознавать идущего человека в режиме реального времени или назвать имя каждого человека находящегося на экране мобильного телефона. Но несмотря на все достижения уровень развития таких систем сопоставим с развитием двухлетнего ребёнка, хотя это уже значительно облегчает работу человека. Компьютерное зрение используется сейчас в широком спектре применяемых для решения каких-либо задач приложениях, которые включают:

- *Оптическое распознавание символов (Optical character recognition или OCR)*. Чтение рукописных почтовых кодов на письмах и распознавание номерных знаков;

- *Проверка оборудования*. Контроль деталей для обеспечения качества с использованием стереофонического зрения со специализированным освещением для измерения допустимых дефектов на крыльях самолетов или автомобильных частях кузова, поиска неисправностей на стальных отливках с использованием рентгеновского зрения;
- *Наблюдение*. Мониторинг нарушителей, которые превышают скорость на дороге или поиск утопающих в бассейнах;
- *Распознавание отпечатков пальцев и биометрия*;
- И др.

Как видно, тема КЗ очень актуальна в современном мире и содержит множество проблем, которые решаются с созданием новых алгоритмов и оптимизацией существующих. Задачей данной дипломной работы является написание web-приложения, которое будет использовать алгоритмы КЗ и распознавать штрих-коды на растровых изображениях. Первая глава посвящена подробному описанию темы компьютерного зрения, описанию алгоритмов, штрих-кодов и их типов, тогда как вторая – построению самого приложения, используя различные технологии.

ГЛАВА 1. АЛГОРИТМЫ КОМПЬЮТЕРНОГО ЗРЕНИЯ. ШТРИХ-КОДЫ

1.1 Алгоритмы компьютерного зрения

1.1.1 Понятие компьютерного зрения

Люди используют свои глаза и свой мозг, чтобы видеть и визуально ощущать мир вокруг себя. Компьютерное зрение - это наука, которая стремится дать похожую, если не лучшую, возможность для машины или компьютера. КЗ связано с автоматическим извлечением, анализом и пониманием происходящей ситуации из одного изображения или из видеопотока. Это предполагает разработку теоретической и алгоритмической основы для достижения автоматического визуального понимания происходящих процессов. Приложения компьютерного зрения многочисленны и используются в таких сферах как:

- сельское хозяйство;
- дополненная реальность;
- автономные транспортные средства;
- биометрия;
- судебно-медицинская экспертиза;
- инспекция качества промышленной продукции;
- анализ жестов;
- восстановление изображения;
- анализ медицинских изображений;
- мониторинг загрязнения;
- робототехника;
- безопасность и наблюдение за транспортом.

КЗ также может быть описано как дополнение (но не обязательно противоположность) биологическому зрению. Больше всего человечество

преуспело в «переизобретении» глаза. За последние несколько лет удалось создать довольно умные системы, которые в зависимости от ситуации могут превосходить по возможностям распознавания даже человека. Благодаря постоянному совершенствованию разработки линз камеры стали очень мощными и способными фокусироваться на интересующих нас объектах на нанометрическом уровне. Кроме того, виден постоянный рост вычислительной мощности, чувствительности датчиков (рисунок 1.1) и камеры теперь могут записывать тысячи изображений в секунду, что требует оптимизации алгоритмов по распознаванию объектов и обработке изображений.

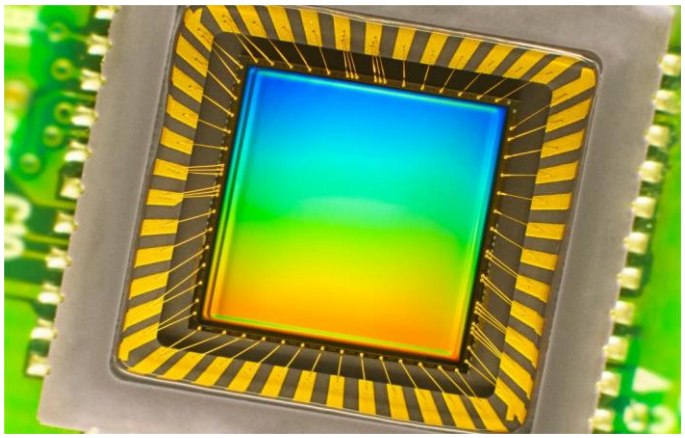

Рисунок 1.1
Данный датчик изображения есть в каждой камере

Большая часть мозга живых существ используется именно для зрения и этот процесс происходит даже на клеточном уровне. Миллиарды клеток работают сообща, чтобы выделить какие-то образцы из необработанного сигнала, поступающего от сетчатки глаза. Если в нем есть какая-то контрастная линия под определенным углом или быстрое движение в каком-то направлении, нейроны начинают двигаться. Сети более высокого уровня преобразуют распознанные образцы в метаобразцы: например, «круглый объект», «движение вверх». К работе подключается следующая по цепочке сеть: «круг белый с красными линиями».

«Объект увеличивается в размерах». На основе этих простых описаний, которые дополняют друг друга, складывается целая картина происходящего.

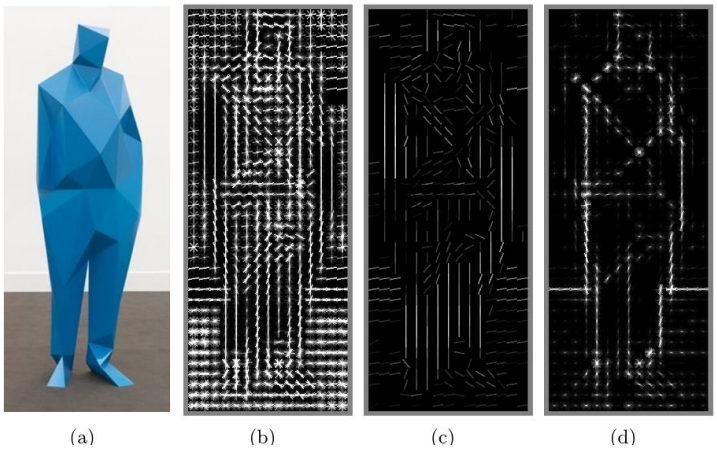

Рисунок 1.2
«Гистограмма направленных градиентов» находит грани и прочие параметры, работая по такому же принципу, как и области мозга, отвечающие за зрение.

Такой метод построения взаимосвязей называется «снизу вверх». Он оказался очень эффективным. С его помощью компьютер может делать ряд трансформаций над изображением, распознавать его края, содержащиеся на нём объекты и многое другое. Все эти процессы происходят благодаря множеству расчетов и статистических вычислений. Сейчас программисты, занятые в сфере КЗ, работают над тем, чтобы смартфоны и другие мобильные устройства могли мгновенно распознавать объекты, которые находятся в поле зрения камеры и накладывать на них текстовое описание. На снимке ниже можно увидеть панораму улицы, которая обработана «железом» с процессором в 120 раз быстрее, чем обычный процессор мобильного телефона.

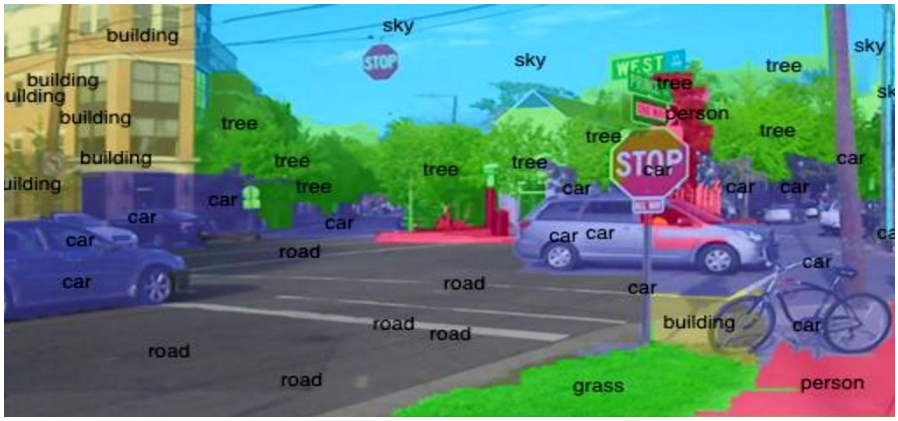

Рисунок 1.3
На данной картинке компьютер распознал и выделил различные объекты, основываясь на известных ему примерах.

Не так давно использовать нейросети было совершенно не практично, т.к. для этого требовалось огромное количество расчетов, что наносит существенный удар по производительности. Но развитие параллельной обработки данных привели к росту исследований и применения систем, пытающихся имитировать работу человеческого мозга. Можно создать компьютерную систему, которая будет распознавать любые сорта яблок — вне зависимости от того, под каким углом они показаны, находятся в движении или в состоянии покоя, целые или надкушенные. Однако, такая система не распознает апельсин. Она даже не сможет сказать, что такое яблоко, съедобный ли это продукт, какого оно размера и для чего оно нужно. А существует проблема эта из-за того, что любому оборудованию нужна операционная система. Для человека такой операционной системой являются отдельные части головного мозга: они обрабатывают информацию, поступающую от наших органов чувств, имеют доступ к кратковременной и долгосрочной памяти, взаимодействуют с вниманием и восприятием, а также содержат данные о приобретённом ранее опыте, который человек получает в процессе взаимодействия с окружающей средой. Все они используют методы, являющиеся до сих пор непонятными для нас. Специалисты в области компьютерных наук, инженеры, психологи, неврологи и философы - они вряд ли могут описать, как работает наш

мозг. Тем не менее компьютерное зрение является полезным даже на таком этапе развития. С его помощью компьютер распознает лица и улыбки на них. Оно помогает беспилотным машинам распознавать дорожные знаки и пешеходов, позволяет промышленным роботам перемещаться между людьми на работе и преодолевать различные препятствия. До момента, когда машины научатся видеть, как человек, пройдёт ещё немало времени, но тот факт, что они уже помогают нам в повседневной жизни, используя КЗ, является большим достижением.

1.1.2 Задачи компьютерного зрения

Каждая из областей применения, описанных выше, использует целый ряд задач компьютерного зрения. Они включают методы получения, обработки, анализа и понимания цифровых изображений, а также извлечения многомерных данных из реального мира с целью получения числовой или символьной информации для последующей обработки. Это подразумевает под собой трансформацию входного изображения, которое преобразует его в массив описаний мира, который затем используется другими мыслительными процессами, т.е. позволяет выделять символическую информацию из картинки, используя модели построенные с помощью геометрии, физики, статистики и др.

1) Распознавание. Классическая проблема компьютерного зрения, обработки изображений и машинного зрения заключается в определении того, содержит ли изображение какой-либо конкретный объект, особенность или активность. Существует несколько направлений проблемы распознавания:

- *распознавание объектов (также называемое классификацией объектов)* - один или несколько предварительно определенных и изученных объектов или классов объектов могут быть распознаны, как правило, вместе с их 2D-позициями в изображении или 3D-позами в сцене;

- *идентификация* - распознается отдельный экземпляр объекта. Примеры включают идентификацию конкретного лица или отпечатка пальца, идентификацию рукописных цифр или конкретного транспортного средства;
- *обнаружение* - изображение сканируется для поиска определённых условий. Примеры включают обнаружение возможных аномальных клеток или тканей на медицинских изображениях или обнаружение транспортного средства в автоматической системе взимания дорожных сборов;
- *Поиск изображений на основе контента* - поиск всех изображений, которые находятся в каком-либо хранилище и имеют конкретный контент. Контент может быть описан различными способами, например, относительно целевого изображения (вернуть все изображения, похожие на искомое изображение), или с помощью критериев более высокого уровня поиска, заданных в текстовом формате (вернуть все изображения, которые содержат много домов, сделанные зимой, и на них нет автомобилей);
- *Оптическое распознавание символов (OCR)* - идентификация символов на изображениях, где есть печатный или рукописный текст;
- *Распознавание лица*;
- *Чтение 2D кодов* – чтение 2D штрих-кодов, таких как матрица данных и QR-кодов;
- *Технология распознавания образов (Shape Recognition Technology or SRT)* – используется в системах учёта людей, отличающих их от объектов (по модели головы и плечам).

2) *Реконструкция сцены*. 3D-реконструкция из нескольких изображений - это создание трехмерных моделей из набора изображений, что является обратным процессом получения 2D-изображений из 3D-сцен. Сама сущность изображения - это проекция из трехмерной сцены на двумерную плоскость, в ходе которой

глубина теряется. Трехмерная точка, соответствующая определенной точке изображения, должна находиться на прямой видимости;

3) *Восстановление изображения.* Целью данной задачи является устранение шума на изображении (шума датчика, размытости изображения и т.д.). Самый простой способ удаления шума – это использование различных фильтров, таких как фильтры нижних частот или медианные фильтры. Более сложные методы предполагают наличие модели того, как выглядит структура исходного изображения, а также модели, отличающей исходную картинку от шума;

4) *Анализ движения.* Анализ движения используется в компьютерном зрении, обработке изображений, высокоскоростной фотографии и машинном зрении, который изучает методы, где обрабатываются два или более последовательных изображения, например, полученных с помощью видеокамеры или высокоскоростной камеры для получения информации на основе распознанного движения в изображениях.

1.1.3 Алгоритмы компьютерного зрения

Информация, приведённая в данном пункте, была получена после прочтения статьи, указанной в «СПИСКЕ ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ» под номером 8.

Существует множество разнообразных методов для распознавания объектов на изображениях. Далее будут описаны три наиболее известных – это контурный анализ, сопоставление по ключевым точкам (feature detection), поиск шаблона (template matching).

Контурный анализ. Данный вариант представляет из себя метод описания, распознавания, хранения, поиска объектов, сравнения по их контурам. Контур – это кривая, которая описывает границу объекта на изображении. Предполагается, что контур содержит достаточно информации о форме объекта, а его внутренние точки не учитываются. Рассмотрение только контуров объектов позволяет уйти от

пространства изображения к пространству контуров, что существенно снижает сложность алгоритмов и вычислений. Главным достоинством контурного анализа является инвариантность относительно вращения, масштаба и смещения контура на тестируемом изображении. Он отлично подходит для поиска объекта некоторой заданной формы. Однако существуют проблемы, связанные с данным методом. Эти проблемы появляются, когда объект имеет одинаковую яркость с фоном изображения, в следствии чего он может не иметь чёткой границы. Также, когда он зашумлён помехами, что приводит к невозможности выделения контура. Возможно перекрытие объектов или их группировка, а это приводит к тому, что контур выделяется неправильно и не соответствует границе объекта.

Таким образом, контурный анализ имеет довольно слабую устойчивость к помехам, и любое нарушение целостности контура или плохая видимость объекта приводят либо к невозможности детектирования, либо к ложным срабатываниям. Однако простота и быстродействие контурного анализа, позволяют вполне успешно применять данный подход при условии наличия чётко выраженного объекта на контрастном фоне и отсутствии помех. Одним из примеров использования контурного анализа является распознание печатного текста. Например, следующий текст, представленный на рисунке 1.4:

Рисунок 1.4
Распознанный текст

вполне будет найден на таком изображении (рисунок 1.5):

Рисунок 1.5
Текст для распознавания

*Template matching***.** Данный метод применяется для поиска участков изображений, которые наиболее схожи с некоторым заданным шаблоном. Таким образом, входными параметрами метода являются:

- изображение, на котором мы будем искать шаблон;
- изображение объекта, который мы хотим найти на тестируемой картинке; размер шаблона должен быть меньше размера проверяемого изображения.

Цель работы алгоритма — найти на тестируемой картинке область, которая лучше всего совпадает с шаблоном. Поиск шаблона производится путем последовательного перемещения его на один пиксель за раз по тестируемому изображению, и оценкой схожести каждой новой области с шаблоном. По результатам проверки выбирается та область, которая имеет наивысший коэффициент совпадения. По сути — это процент совпадения области картинки и шаблона. Template matching является хорошим выбором, когда необходимо быстро проверить наличие некоторого объекта на изображении. Также в интернете можно найти различные примеры использования алгоритма для идентификации человека по лицу. Такой подход значительно проще реализовать, нежели при помощи обучаемых алгоритмов. Однако стоит понимать, что template matching не позволяет с уверенностью сказать был ли найден исходный объект, поскольку это вероятностная характеристика, зависящая от масштаба, углов обзора, поворотов картинки и наличия физических помех. Также возможны ложные срабатывания

алгоритма, когда искомого объекта на самом деле нет, но имеются какие-то общие детали у шаблона и области на тестируемом изображении. Конечно, подобной ситуации можно избежать путём проверки значения коэффициента совпадения (чтобы он не был меньше некоторого граничного предела), однако это не всегда будет работать должным образом ввиду описанных выше причин.

*Feature Detection***.** Концепция feature detection в компьютерном зрении относится к методам, которые нацелены на вычисление абстракций изображения и выделения на нем ключевых особенностей. Данные особенности затем используются для сравнения двух изображений с целью выявления у них общих составляющих. Не существует строгого определения того, что такое ключевая особенность картинки. Ею могут быть как изолированные точки, так и кривые или некоторые связанные области. Примерами таких особенностей могут служить грани объектов и углы. В отличие от template matching и контурного анализа, алгоритмы поиска ключевых точек более устойчивы к помехам, трансформациям и позволяют находить объекты даже при наличии физических помех. При этом высокая скорость работы некоторых методов позволяет применять их для поиска изображений в режиме реального времени даже на мобильных устройствах, что привело к возможности использования дополненной реальности в смартфонах и планшетах**.**

1.2 Штрих-коды

1.2.1 Понятие штрих-кода

Штриховой код (часто обозначаемый одним словом штрих-код) – это небольшое изображение линий (баров) и пробелов между ними, которое нанесено на различные продовольственные товары, идентификационные карточки, почтовые сообщения, чтобы идентифицировать продукт, человека или локацию. Код использует последовательность вертикальных линий (баров) и пробелов между

ними представляя тем самым числа или другие символы. Значение, которое представляет штрих-код состоит из пяти частей:

- Префикс национальной организации GS1 (3 цифры);
- Регистрационный номер производителя товара (4-6 цифр);
- Код товара (3-5 цифр);
- Контрольное число (1 цифра);
- Дополнительное поле (необязательное штрихкодовое поле, иногда там ставится знак «>», «индикатор свободной зоны»).

Обычно ШК представляют различные данные путём изменения ширины и расстояния между вертикальными линиями и называются линейными или одномерными (1D). Также существуют двумерные ШК, которые используют прямоугольники, точки, шестиугольники и другие геометрические фигуры. Такие типы кодов тоже называются штрих-коды, хотя они не используют баров как таковых.

ШК первоначально сканировались специальными оптическими сканерами, которые называются считыватели штрих-кодов. Позднее стало доступно программное обеспечение для устройств, которые могли читать изображения, такие как смартфоны с камерами, планшеты и др.

1.2.2 Использование штрих-кодов

Штрих-коды, как правило, используются для отслеживания инвентаризации, но есть еще много возможностей и интересных способов их использования:

- *Мероприятия, путешествия, фильмы.* Штрих-коды используются в авиационных билетах (рисунок 1.6), во входных билетах в кинотеатрах и других мероприятиях, чтобы однозначно идентифицировать и проверить действительность билета до того, как клиент сможет войти в театр или на мероприятие. Они также используются для подсчета продаж, полученных на мероприятии, и делают его

намного более удобным для отслеживания доходов. Иногда это помогает уменьшить расходы для организаторов мероприятий, поскольку им не нужно тратить деньги на изготовление реальных билетов - те, кто хочет принять участие в мероприятии или посмотреть шоу, могут распечатать штрих-код в любом месте и представить его по прибытии на место проведения.

Рисунок 1.6
Пример использования штрих-кодов в авиационных билетах

Аналогичным образом, с использованием систем онлайн-бронирования для поездок (полеты, поезда и автобусы), путешественники могут распечатать собственный посадочный талон и зарегистрироваться, показав код с помощью своего смартфона.

- *Реклама*. Рекламодатели пользуются штрих-кодами (рисунок 1.7), чтобы обращаться к клиентам более интерактивным, интересным и уникальным способом. С помощью смартфонов, просто загрузив и установив приложение, которое может читать штрих-коды, потенциальный покупатель может узнать гораздо больше информации о рекламируемом продукте.

Рисунок 1.7
Пример использования штрих-кода рекламодателями

- На изобоажении выше присутствует 2D-штри-код. Его отличие от 1D штрих-кодов состоит в том, что он способен хранить больше информации (и языков). Эти штрих-коды обычно содержат унифицированные указатели ресурсов (URL), чтобы человек мог больше узнать о продукте или услуге. Хотя такой способ предоставления информации не очень популярен в Беларуси, использование 2D-штрих-кодов применяется во многих сферах жизни в Японии, где они и были изобретены. После землетрясения 2011 года и цунами QR-коды были особенно полезны в распространении информации о том, как оказать первую медицинскую помощь пострадавшим в бедствии (рисунок 1.8).

Рисунок 1.8
как оказать первую медицинскую помощь пострадавшим в бедствии

- *Отслеживание потребляемой пищи (рисунок 1.9).*

Рисунок 1.9
Отслеживание потребляемой пищи

Существует множество приложений, которые помогают человеку получать подробную информацию о потребляемой им пищи. Это может приносить пользу, когда ведётся дневник питания и отслеживание потребляемых калорий.

- И др.

Кроме полезных функций штрих-кодов, могут встречаться оригинальные идеи, которые не несут какого-либо смысла. К примеру, искусство и поп-культура (рисунок 1.10):

Рисунок 1.10
Санкт-Петербург, Россия

Штрих-коды стали настолько популярными, что они были использованы в построении архитектуры современного здания и замечены в кадрах популярных фильмов (рисунок 1.11).

Рисунок 1.11
На автомобили в фильме «Назад в будущее» (Universal, 1985), вместо номерных знаков используются штрих-коды

В заключении первой главы можно сделать следующий вывод: задача компьютерного зрения актуальна на сегодняшний день, т.к. компьютер помогает решать человеку огромное количество задач в разных сферах жизни, включая работу и развлечения. Возможность машины распознавать объекты и анализировать ситуацию на изображении позволит ей расширить спектр своего применения. Следовательно, нужно разрабатывать новые алгоритмы и улучшать существующие, т.к. многие из них требуют тщательной доработки и анализа проблем, появляющихся на границах корректной работы программы. Также можно видеть, что штрих-коды являются важным объектом, который стоит рассмотреть в сфере КЗ. Большое распространение ШК в окружающем нас мире требует наличия алгоритмов, которые могут быстро и качественно распознавать их на изображении, сделанном с помощью камеры мобильного телефона, планшета и даже умных часов, а это значит, что нужно учесть и проблему потребляемых программой ресурсов устройства, на которых она будет использоваться.

1.2.3 Типы штрих-кодов

На сегодняшний день известно большое разнообразие типов штрих-кодов, которые делятся на две общие группы – это линейные (одномерные) и двумерные штрих-коды. Линейные штрихкоды представляют собой набор полосок, расположенных в одном направлении. Стандарты линейных штрихкодов:

- EAN – европейский (EAN-8 состоит из 8 цифр, EAN-13 — используются 13 цифр). Штрих-коды EAN используются для маркировки потребительских товаров прежде всего в Европе. Они очень похожи на коды UPC, а основное отличие - их географическое применение. EAN-13 (состоящий из 13 цифр) является более распространённым, чем EAN-8 (состоящий из 8 цифр). Последний можно встретить на продуктах, где доступно ограниченное пространство, например небольшие конфеты;
- UPC (UPC-A, UPC-E). Штрих-коды UPC используются для маркировки и сканирования потребительских товаров по всему миру, главным образом в США, но также в Великобритании, Австралии, Новой Зеландии и других странах. Вариант UPC-A кодирует 12 цифр, в то время как UPC-E является меньшим вариантом, который кодирует только 6 цифр;
- Code39. Коды Code39 (или Code 3 of 9) используются для маркировки товаров во многих отраслях производства и часто встречаются в автомобильной промышленности и в Министерстве обороны США. Он позволяет использовать как цифры, так и символы, а называется так из-за того, что может кодировать только 39 символов, хотя в его последней версии набор символов был увеличен до 43;
- Code128. Code 128 штрих-кодов - это компактные, высокоплотные коды, используемые в логистических и транспортных. Могут хранить разнообразную информацию, поскольку они поддерживают любой символ из набора ASCII 128;

- Codabar. Штрих-коды Codabar используются специалистами по логистике и здравоохранению, включая банки крови в США, фотолаборатории и библиотеки. Его главным преимуществом является то, что он может легко печататься на любом принтере и даже на пишущей машинке. Кодирует до 16 различных символов с дополнительными 4 символами начала / остановки.

Такие ШК могут нести уникальную зашифрованную информацию о товаре (до 30 символов). Линейные штрихкоды различаются по плотности. Стандарты 2D штрих-кодов:

- Aztec Code. Коды Aztec - это 2D-штрих-коды, используемые транспортной отраслью, в частности, для билетов авиакомпаний. Данные штрих-коды могут быть декодированы, даже если они имеют плохое разрешение, что делает их полезными, когда билеты печатаются плохо и когда они представлены на телефоне. Кроме того, они могут занимать меньше места, чем другие штрих-коды, потому что им не требуется окружающая пустая «тихая зона», в отличие от некоторых других типов 2D-штрих-кодов;
- Data Matrix. Данные коды используются для обозначения мелких предметов, товаров и документов. Маленький размер делает их идеальными для маркировки небольших продуктов. Подобно QR-кодам, они имеют высокую отказоустойчивость и быструю читаемость;
- PDF417. Это 2D-штрих-коды, используемые в приложениях, требующих хранения больших объемов данных, таких как фотографии, отпечатки пальцев, электронные подписи, текст, цифры и графика. Они могут хранить более 1,1 килобайта данных, что делает их намного более мощными, чем другие 2D-штрих-коды. Как и QR-коды, штрих-коды PDF417 являются общедоступными и бесплатными;
- QR код. Это 2D-матричные штрих-коды, которые часто используются в рекламных объявлениях, журналах и визитных карточках. Свободные в

использовании, гибкие по размеру, имеют высокую отказоустойчивость и быструю читаемость, хотя их невозможно прочитать с помощью лазерного сканера. QR-коды поддерживают разные режимы кодирования данных: числовые, буквенно-цифровые, байтовые / двоичные.

Преимуществами 2D штрих-кодов можно считать большую информативность. Вместо максимальных 30 символов линейного, можно зашифровать целое послание на 2000 символов и более.

ГЛАВА 2. РАЗРАБОТКА ПРИЛОЖЕНИЯ

2.1 Функциональные требования к дипломной работе

Одной из задач дипломной работы было создание веб-приложения, которое можно открыть как на компьютере, так и на мобильном телефоне или планшете. Функционал данного приложения должен позволять загружать изображение из файловой системы либо делать снимок с камеры устройства для дальнейшей обработки и поиска штрих-кода. В результате работы программы должна быть выделена цветным контуром область, в которой находится штрих-код.

2.2 Описание технологий и архитектуры приложения

Для создания приложения было решено использовать определённый стек технологий, который содержит в себе библиотеки для работы с алгоритмами компьютерного зрения, фреймворки для построения UI части приложения и взаимодействия с ним.

Программа состоит из фронт-энд части, которая реализована с помощью Angular 2 и typescript, сервера, написанного на node.js, скрипта на python и двух библиотек: QuaggaJS – для реализации алгоритма на языке программирования JavaScript и OpenCV - для реализации алгоритма на языке программирования python. Использование Angular 2 вместе с typescript является хорошим решением, т.к. несмотря на свою относительную новизну эти технологии уже получили уважение среди front-end разработчиков, а python – предоставляет лаконичный синтаксис и большое разнообразие встроенных функций. QuaggaJS и OpenCV – одни из самых популярных библиотек для обработки изображений на языках JavaScript и Python соответственно.

Angular является основой для построения клиентских приложений с помощью HTML и JavaScript, либо языка TypeScript, который компилируется в JavaScript. Структура состоит из нескольких библиотек, некоторые из которых являются

основными и некоторые необязательными. Написание приложения на Angular, подразумевает составления HTML-шаблонов, с определённой разметкой, написания классов-компонентов для управления этими шаблонами, добавления логики приложения в сервисы. Этот framework берет на себя все функции по представлению содержимого приложения в браузере и реагируя на взаимодействие с пользователем в соответствии с инструкциями, которые он предоставил. Основная структура любого web-приложения, написанного на Angular framework, представлена на рисунке 2.1.

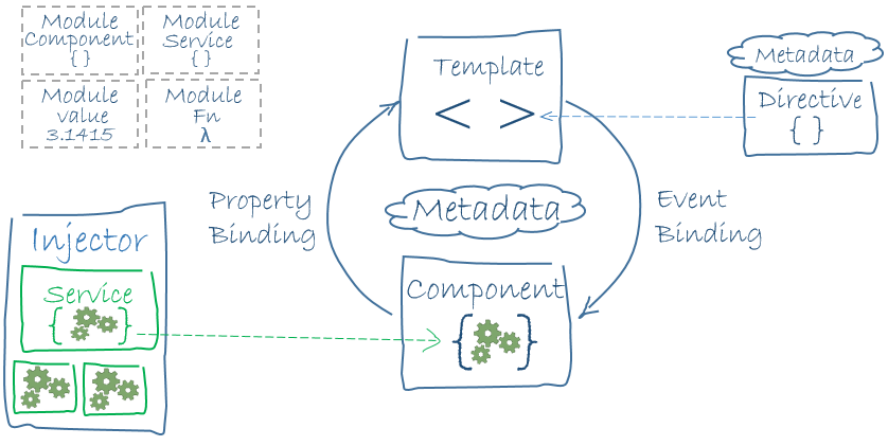

Рисунок 2.1
Основная структура любого web-приложения, написанного на Angular framework

TypeScript - это свободный и открытый язык программирования, разработанный и поддерживаемый Microsoft. Он, по сути, является надмножеством над JavaScript и добавляет к языку необязательную статическую типизацию и объектно-ориентированное программирование на основе классов. Работал над разработкой TypeScript Андерс Хейлсберг, ведущий архитектор C # и создатель Delphi и Turbo Pascal. TypeScript может использоваться для разработки JavaScript-приложений, исполняющихся на стороне клиента или на стороне сервера (Node.js). Поскольку TypeScript является надмножеством над JavaScript, все существующие программы JavaScript также являются совместимыми с программами, написанными на TypeScript.

Node.js - это кросс-платформенная среда с открытым исходным кодом, для выполнения языка JavaScript на стороне сервера. В приложении она используется следующим образом: на сервер поступает запрос обработать изображение. После этого с помощью специального модуля загружается скрипт написанный на Python, который в свою очередь занимается обработкой картинки.

Python - широко используемый язык программирования высокого уровня общего назначения, созданный Guido van Rossum и впервые выпущенный в 1991 году. Интерпретируемый язык Python приносит разработчикам удобочитаемость кода (в частности, использование пробелов для разделения кодовых блоков вместо фигурных скобок или ключевых слов) и синтаксис, который позволяет программистам выражать идеи с меньшим числом строк кода, чем это возможно в таких языках, как C ++ или Java. Этот язык широко используется для написания как небольших, так и крупных проектов.

QuaggaJS - это библиотека для сканирования штрих-кодов, полностью написанная на JavaScript, которая производит локализацию и декодирование различных типов ШК в реальном времени, таких как EAN, CODE 128, CODE 39, EAN 8, UPC-A, UPC-C, I2of5 и CODABAR. Она также может использовать getUserMedia для получения прямого доступа к потоку камеры пользователя. Хотя код подразумевает тяжелую обработку изображений, даже слабые по мощности смартфоны способны обнаруживать и декодировать штрих-коды в режиме реального времени.

OpenCV (Open Source Computer Vision Library) – это библиотека с открытым исходным кодом, которая помогает реализовать множество алгоритмов компьютерного зрения и решает задачи машинного обучения. В библиотеке имеется более 2500 оптимизированных алгоритмов. Эти алгоритмы могут использоваться для обнаружения и распознавания лиц, идентификации объектов, классификации действий человека на видео, отслеживания движущихся объектов, извлечения 3D-моделей объектов, нахождения похожих изображений из базы уже

имеющихся, удаления красных глаз на изображениях, сделанных с помощью вспышки, слежения за движениями глаз и т. д. Сообщество OpenCV насчитывает более 47 тысяч человек и количество загрузок превышает 14 миллионов. Библиотека широко используется в компаниях, исследовательских группах и правительственных органах.

Для запуска приложения нужно зайти в его корневую директорию и выполнить команду **npm start**, после чего в браузере откроется новая вкладка по адресу http://localhost:3000/start-page. Эта страница является стартовой в приложении (рисунок 2.2).

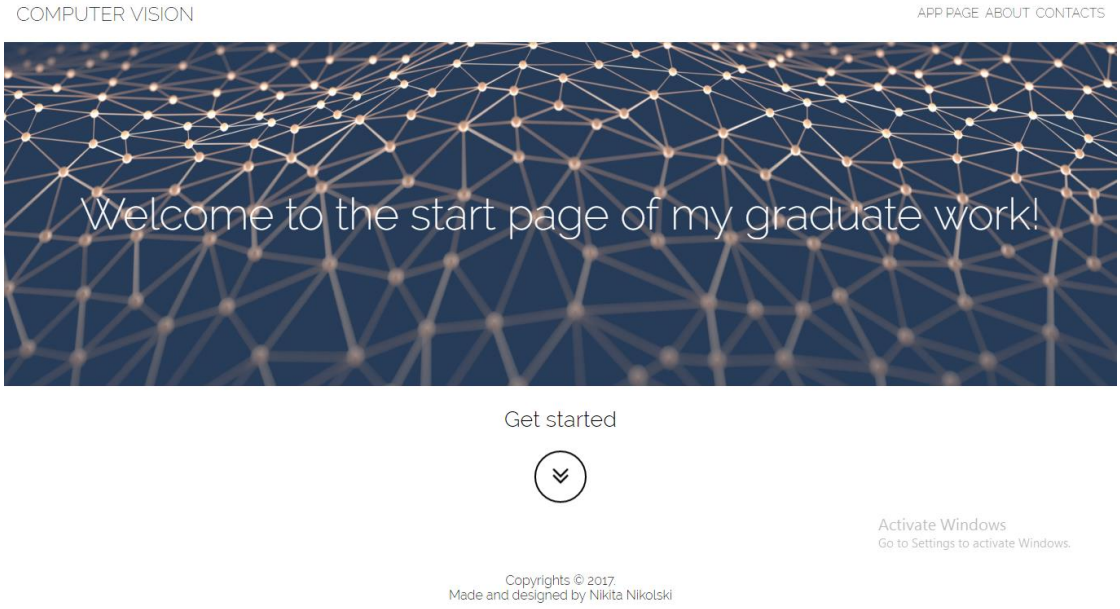

Рисунок 2.2
Стартовая страница

Нажав на кнопку Get started происходит переход на страницу с выбором способа распознавания штрих-кода (рисунок 2.3).

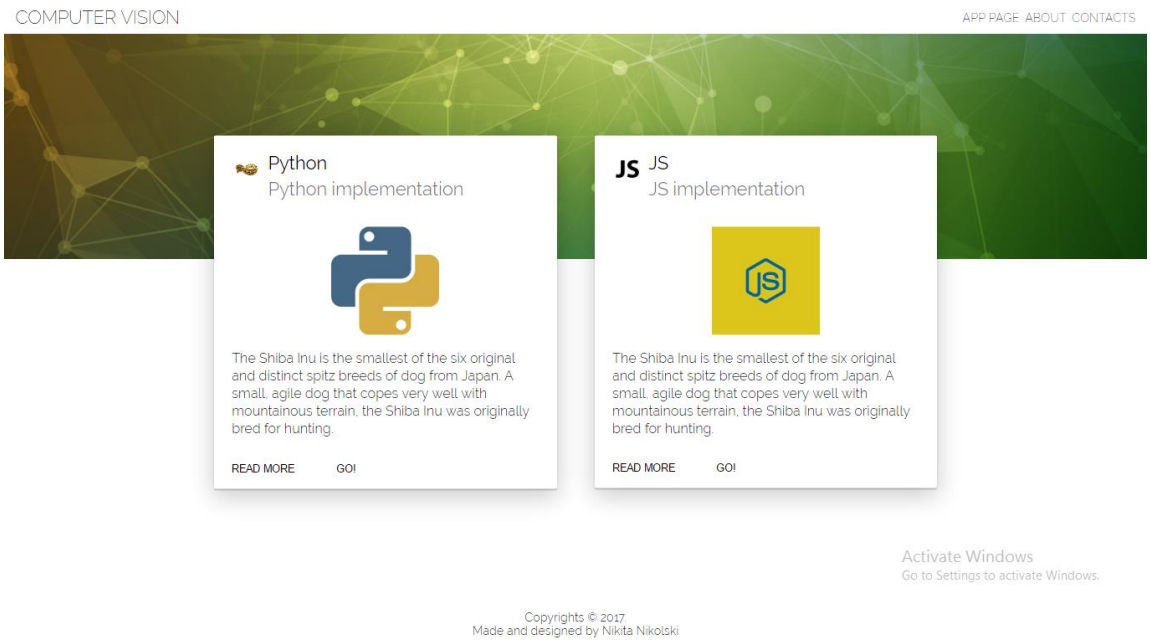

Рисунок 2.3
Страница с выбором способа распознавания штрих-кода

По нажатию на кнопку “READ MORE” происходит переход на сторонний ресурс с подробным описанием соответствующей технологии. Выбрав одну из табличек и нажав на кнопку “GO”, браузер перенаправляет нас на страницу с реализованным функционалом. На странице с реализацией на JavaScript-е есть возможность выбрать один из способов получения изображения: загрузка с файловой системы или распознавание с помощью камеры устройства в режиме реального времени (рисунок 2.4).

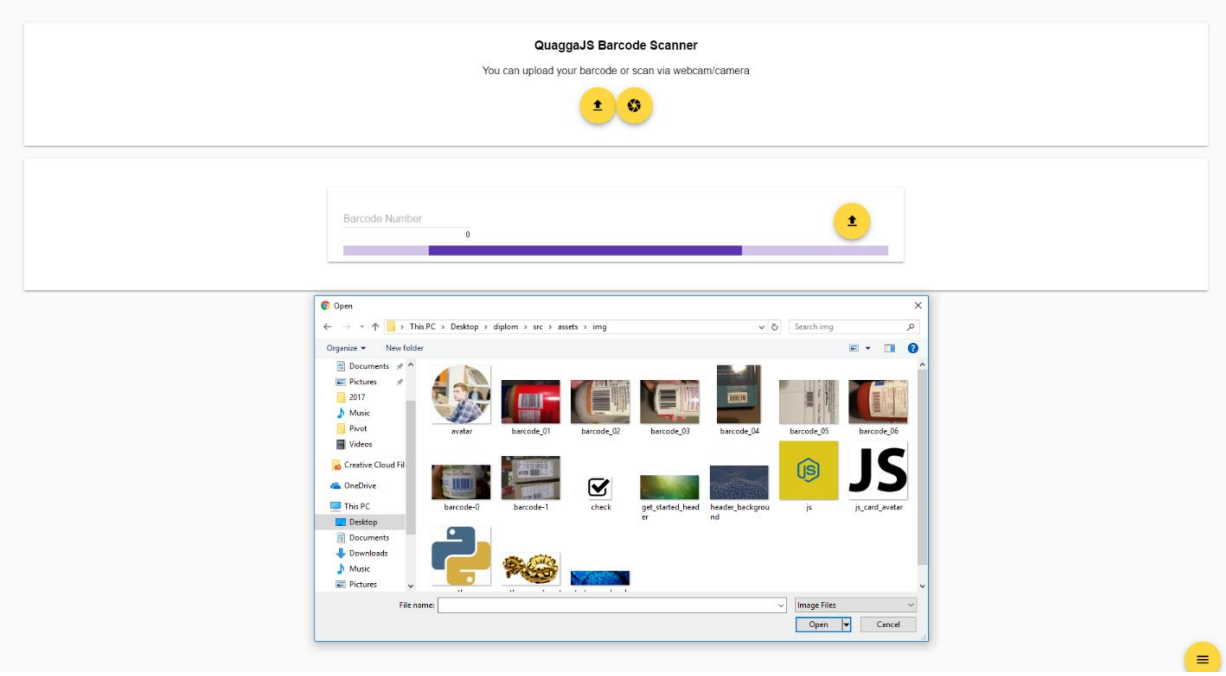

Рисунок 2.4
Страница с JavaScript реализацией

Результат работы данного варианта реализайции можно видеть на рисунке 2.5

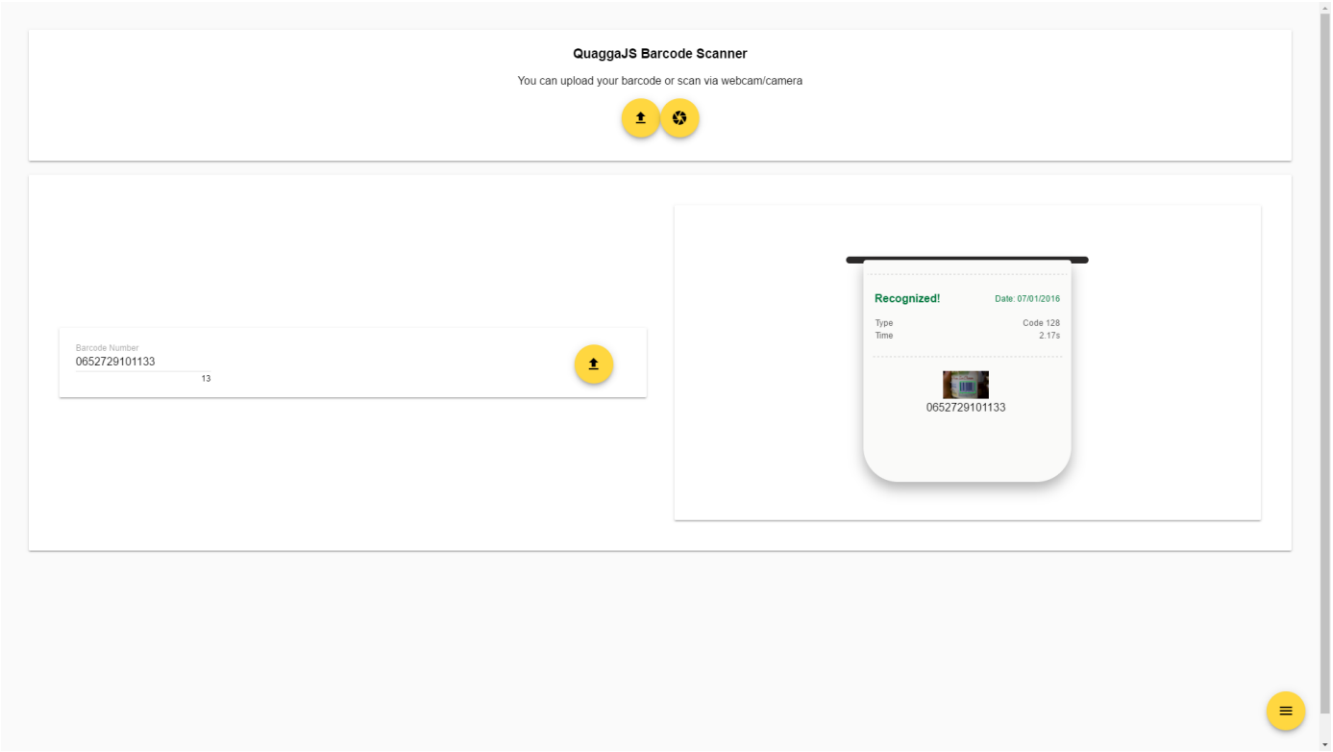

Рисунок 2.5
Результат работы JavaScript реализации

Выбрав табличку с реализацией на python и OpenCV, браузер перенаправляет нас на соответствующую страницу. Здесь есть возможность загрузить картинку с файловой системы. Результат работы данного типа реализации можно видеть на рисунке 2.6.

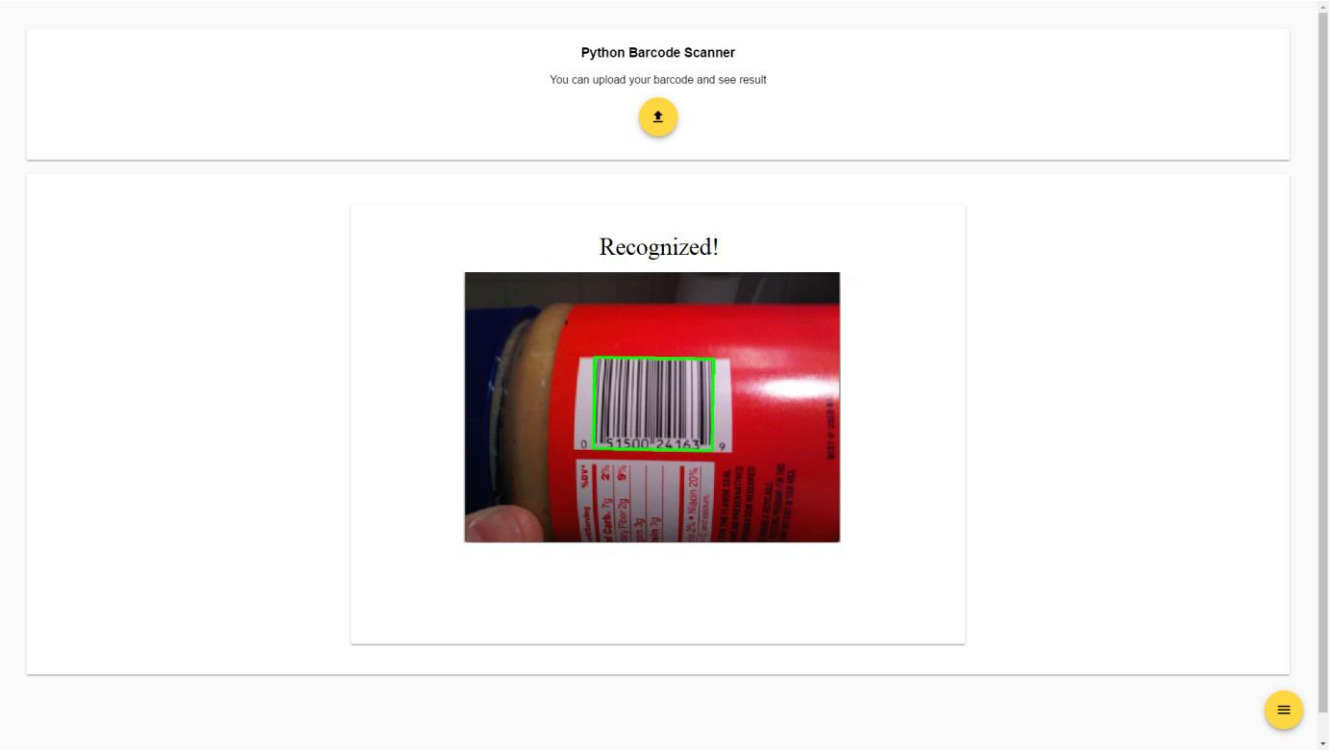

Рисунок 2.6
Результат работы Python реализации

В любом месте приложения вверху справа можно нажать на ссылки для перехода на главную страницу приложения, страницу с описанием дипломной работы и со списком контактов студента, выполнившего её (рисунок 2.7 и рисунок 2.8 соответственно).

Рисунок 2.7
Страница с описанием работы

Рисунок 2.8
Страница с контактами автора дипломной работы

2.3 Использование библиотеки QuaggaJS

2.3.1 Анализ функционала библиотеки

Библиотека QuaggaJS основывается на использовании современного Web-APIs, которые не реализованы во всех браузерах, что не позволяет сказать о полной кроссбраузерности разработанного с её помощью решения. Существует два режима, в которых может работать библиотека:

1) Анализ статических изображений;

2) Использование камеры для обработки изображения в режиме реального времени.

С более подробной информацией о данной библиотеке, её предоставляемой информацией и примерами можно ознакомиться, прочитав статью, указанную в «СПИСКЕ ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ» под номером 6.

Для анализа статических изображений в браузере должны присутствовать следующие APIs: webworkers, canvas, typedarrays, bloburls, blobbuilder. Для распознавания ШК в режиме реального времени нужно также наличие MediaDevices API. Чтобы установить библиотеку необходимо выполнить команду **npm install quagga** в корневой папке проекта. Далее нужно импортировать зависимости в проекте:

```
import Quagga from 'quagga';
```

В настоящее время полноценная функциональность доступна только в браузере. Используя библиотеку на стороне сервера с помощью node.js, будет работать только распознавание изображений, взятых с файловой системы.

Quagga.init(config, callback) – данный метод инициализирует библиотеку с переданной ей конфигурацией (будет описана ниже) и запускает функцию обратного вызова (callback), когда Quagga выполнит стадию начальной загрузки.

Процесс инициализации требует также запроса к доступу камеры, если режим распознавания в реальном времени был указан в конфигурационном объекте. В случае ошибки, в callback передаётся параметр **err**, который содержит информацию о причине её возникновения. Ошибка может случится, когда в параметр **inputStream.type** установлено значение **LiveStream**, но браузер не поддерживает такое API или пользователь просто отклонил разрешение на использование камеры телефона. Ниже приведён пример функции инициализации, в которую передаётся конфигурационный объект и функция обратного вызова:

```
Quagga.init({
  inputStream : {
    name : "Live",
    type : "LiveStream",
    target: document.querySelector('#yourElement')
  },
  decoder : {
    readers : ["code_128_reader"]
  }
}, function(err) {
    if (err) {
        console.log(err);

        return;
    }

    console.log("Initialization finished. Ready to start");

    Quagga.start();
});
```

Quagga.start(): когда библиотека инициализирована, метод **start()** запускает видеопоток и начинает процесс обработки изображения и распознавания штрих-кода.

Quagga.stop(): если декодер в данный момент работает, то после вызова функции **stop()** он больше не будет обрабатывать изображения. Кроме того, если

после инициализации было запрошено разрешение на использование камеры, эта операция также отключает камеру.

Quagga.onProcessed(callback): этот метод регистрирует **callback(data)** функцию, которая будет вызываться после завершения обработки каждого кадра. Объект **data** содержит подробную информацию о результате операции.

Quagga.onDetected(callback) – регистрирует функцию, которая будет запускаться каждый раз, когда штрих-код был найден и успешно распознан. Передаваемый объект **data** содержит информацию о процессе декодирования, включая обнаруженный код, который может быть получен путем вызова data.codeResult.code.

Quagga.decodeSingle(config, callback) – отличие данной функции от предыдущей в том, что данный метод распознавания не полагается на **getUserMedia** и вместо него обрабатывает одно входящее изображение. Передаваемый **callback,** такой же, как и **onDetected** и принимает объект с результатом операции **data.**

Quagga.offProcessed(handler) – в случае, когда событие **onProcessed** больше не актуально, метод **offProcessed** удаляет данный обработчик из очереди событий.

Quagga.offDetected(handler) – в случае, когда событие **onDetected** больше не актуально, метод **offDetected** удаляет данный обработчик из очереди событий.

Функция **callback,** передаваемая в **onProcessed, onDetected, decodeSingle** принимает объект **data** при выполнении. Этот объект может содержать различную информацию. В зависимости от успеха некоторые поля могут быть неопределенными или просто пустыми. Ниже приведён пример объекта **data:**


```
{
 "codeResult": {
  "code": "FANAVF1461710",
  "format": "code_128",
  "start": 355,
  "end": 26,
  "codeset": 100,
```

```
  "startInfo": {
    "error": 1.0000000000000002,
    "code": 104,
    "start": 21,
    "end": 41
  },
  "decodedCodes": [{
    "code": 104,
    "start": 21,
    "end": 41
  },
  {
    "error": 0.8888888888888893,
    "code": 106,
    "start": 328,
    "end": 350
  }],
  "endInfo": {
    "error": 0.8888888888888893,
    "code": 106,
    "start": 328,
    "end": 350
  },
  "direction": -1
},
"line": [{
  "x": 25.97278706156836,
  "y": 360.5616435369468
}, {
  "x": 401.9220519377024,
  "y": 70.87524989906444
}],
"angle": -0.6565217179979483,
"pattern": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, /* ... */ 1],
"box": [
  [77.4074243622672, 410.9288668804402],
  [0.050203235235130705, 310.53619724086366],
  [360.15706727788256, 33.05711026051813],
  [437.5142884049146, 133.44977990009465]
],
"boxes": [
```

```
    [
      [77.4074243622672, 410.9288668804402],
      [0.050203235235130705, 310.53619724086366],
      [360.15706727788256, 33.05711026051813],
      [437.5142884049146, 133.44977990009465]
    ],
    [
      [248.90769330706507, 415.2041489551161],
      [198.9532321622869, 352.62160512937635],
      [339.546160777576, 240.3979259789976],
      [389.5006219223542, 302.98046980473737]
    ]
  ]
}
```

Конфигурация по умолчанию описывает простой вариант использования библиотеки, но это не мешает её удобно настроить для каких-то конкретных целей. Настройка происходит посредством объекта **config,** который может определять следующие свойства:

```
{
  numOfWorkers: 4,
  locate: true,
  inputStream: {...},
  frequency: 10,
  decoder:{...},
  locator: {...},
  debug: false,
}
```

numOfWorkers – QuaggaJS поддерживает технологию web workers из коробки и работает с четырьмя workers по умолчанию. Их число должно совпадать с количеством ядер, доступных на исполняемом устройстве. Web worker - это JavaScript, который работает в фоновом режиме независимо от других скриптов, не влияя на производительность страницы. При выполнении скриптов на HTML-странице она перестает отвечать на запросы, пока скрипт не будет завершен. Таким образом с помощью данной технологии пользователь может продолжать делать то,

что хочет: кликать, выбрать текст и т. д., в то время как web workers работают в фоновом режиме.

locate - одной из основных особенностей QuaggaJS является способность находить штрих-код на входном изображении. Свойство locate определяет, включена ли эта функция (по умолчанию) или выключена. Нахождение местоположения ШК является дорогостоящей в плане вычисления операцией и иногда данный процесс может некорректно работать на некоторых устройствах. Отключение данной опции является полезным в тех случаях, когда ориентация или приблизительная позиция штрих-кода известны, или если требуется от пользователя вручную выделить штрих-код прямоугольным контуром. Это может одновременно повысить производительность и надежность.

inputStream – определяет источник изображений или видео.

```
{
  name: "Live",
  type: "LiveStream",
  constraints: {
    width: 640,
    height: 480,
    facingMode: "environment",
    deviceId: "7832475934759384534"
  },
  area: { // определяет прямоугольник для обнаружения в области
    top: "0%",
    right: "0%",
    left: "0%",
    bottom: "0%"
  }
}
```

Свойство **type** может содержать три разных значения: **ImageStream**, **VideoStream** или **LiveStream** (по умолчанию).

Объект **constraints** определяет физические размеры входного изображения и дополнительные свойства, такие как **faceMode**, который устанавливает

источник камеры пользователя в случае нескольких подключенных устройств. Объект **area** ограничивает область декодирования изображения. Значения устанавливаются в процентах. Это свойство также полезно в случае, когда **locate** установлено в **false**.

frequency - это свойство управляет частотой сканирования видеопотока. Оно необязательно и определяет максимальное число проверок в секунду, чтобы облегчить работу процессора.

decoder - QuaggaJS обычно используется в двухэтапном режиме: **locate** устанавливается в **true**, а после того, как штрих-код обнаружен, начинается процесс декодирования - преобразование баров в значение, стоящее за ними. Большинство параметров конфигурации в декодере предназначены только для целей отладки или визуализации.

```
{
  readers: [
    'code_128_reader'
  ],
  debug: {
      drawBoundingBox: false,
      showFrequency: false,
      drawScanline: false,
      showPattern: false
  }
  multiple: false
}
```

Наиболее важным свойством является **readers**, которое принимает массив типов штрих-кодов, которые должны быть декодированы. Возможные значения:

- *code_128_reader (default)*
- *ean_reader*
- *ean_8_reader*
- *code_39_reader*

- *code_39_vin_reader*
- *codabar_reader*
- *upc_reader*
- *upc_e_reader*
- *i2of5_reader*

Свойство **multiple** сообщает декодеру, должен ли он продолжить декодирование после нахождения нужного типа ШК. Если для параметра **multiple** установлено значение **true**, результаты будут возвращены в виде массива объектов. Каждый объект в массиве будет иметь ограничивающую рамку и может иметь значение **codeResult**, зависящее от успешного декодирования ШК.

locator – наличие данного объекта имеет значение только в том случае, если флаг **locate** установлен в true. Он управляет поведением процесса локализации и должен быть настроен. Здесь важны только два свойства для использования в **Quagga - halfSample** и **patchSize**, тогда как остальные нужны только для разработки и отладки.

```
{
    halfSample: true,
    patchSize: "medium", // x-small, small, medium, large, x-large
    debug: {
      showCanvas: false,
      showPatches: false,
      showFoundPatches: false,
      showSkeleton: false,
      showLabels: false,
      showPatchLabels: false,
      showRemainingPatchLabels: false,
      boxFromPatches: {
        showTransformed: false,
        showTransformedBox: false,
```

```
      showBB: false
    }
  }
}
```

Флаг **halfSample** указывает процессу-локатору, следует ли ему работать с уменьшенным изображением (половина ширины / высота, четверть пикселей) или нет. Включение **halfSample** позволяет значительно сократить время обработки, а также помогает найти шаблон штрих-кода из-за неявного сглаживания. Он должен быть отключен в случаях, когда штрих-код действительно мал, и для определения позиции требуется большее разрешение. Второе свойство **patchSize** определяет плотность сетки поиска. Свойство принимает строки со значением **x-small**, **small**, **medium**, **large** и **x-large**. Если на изображении присутствуют штрих-коды большого относительно других объектов размера, то рекомендуется использовать **large** или **x-large**. В случаях, когда штрих-код находится дальше от объектива камеры, рекомендуется установить размер **small** или даже **x-small**. В последнем случае также рекомендуется выставить разрешение, чтобы найти штрих-код.

В следующем примере принимается адрес изображения **src** и выводится результат в консоли:

```
Quagga.decodeSingle({
    decoder: {
        readers: ["code_128_reader"] // список типов ШК
    },
    locate: true,
    src: '/test/fixtures/code_128/image-001.jpg'
}, function(result){
    if(result.codeResult) {
        console.log("result", result.codeResult.code);
    } else {
        console.log("not detected");
```

```
    }
});
```

2.3.2 Анализ работы встроенных алгоритмов

Информация, приведённая в этом пункте, получена после прочтения электронного ресурса, а именно статьи, указанной в «СПИСКЕ ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ» под номером 7.

Главная цель локатора штрих-кода - найти образец внутри изображения, который выглядит как штрих-код. Штрих-код обычно характеризуется черными полосами и пробелами между ними. Общий размер штрих-кода зависит от количества закодированной информации (Code128). Однако, штрих-код также может быть зафиксирован по ширине (EAN13).

Когда стоит задача найти штрих-код на изображении, обычно ищут линии, которые близко расположены друг к другу и имеют одинаковый угол наклона. Алгоритм распознавания следующий:

1) Создание двоичного представления изображения
2) Построить сетку изображения (20 x 15 ячеек)
3) Извлечь скелет каждой ячейки
4) Разделить скелет на части
5) Маркировка компонентов
6) Рассчитать угол наклона каждой части
7) Определение качества ячейки (один угол наклона компонентов)

*1) Создание двоичного представления изображения***.** Существует много разных способов создания двоичного представления изображения. Самый простой способ - установить глобальный порог 127 и определить для каждого пикселя его более яркий (> = 127) или более темный (<127) спектр, чем заданный порог. Хотя этот метод может быть простым в реализации, иногда результат может не удовлетворять, потому что он не учитывает изменения яркости на изображении.

Чтобы учесть это, был выбран метод Otsu, который работает на гистограмме исходного изображения и пытается отделить передний план от фона. Он является полностью автоматизированным и основывается на анализе распределения яркости пикселей изображения по его нормализованной гистограмме:

$$\mathbf{p_i = n_i / N} \tag{2.1}$$

где **N** – общее число пикселей изображения, $\mathbf{n_i}$ – число пикселей с уровнем яркости **i**, **i = 0 .. L**. Метод позволяет разделить изображение на 2 класса относительно граничного значения t, где класс c_1 содержит пиксели с яркостями **[0, 1, ... , t]**, а класс $\mathbf{c_2}$ – пиксели с яркостями **[t+1, ... , L–1]**. Поиск граничного значения основан на минимизации внутриклассового различия, представленного в виде суммы различий каждого кластера:

$$\mathbf{\sigma_w^2 (t) = P_1(t) \sigma_1^2 (t) + P_2(t) \sigma_2^2 (t)} \tag{2.2}$$

Веса $\mathbf{P_i}$ – это вероятности класса **i**, $\mathbf{\sigma_i^2}$ – внутриклассовые различия. Можно учитывать максимизацию межклассовых различий:

$$\mathbf{\sigma_b^2 (t) = \sigma^2 – \sigma_w^2 (t) = P_1(t)P_2(t)[\mu_1(t) - \mu_2(t)]^2} \tag{2.3}$$

где $\mathbf{\sigma^2}$ – совокупная дисперсия. Вероятность класса по данному порогу вычисляется по следующей формуле:

$$\mathbf{P_1(t) = \sum_{i=0}^{t} p(i), P_2(t) = \sum_{i=t+1}^{L-1} p(i) = 1 – P_1} \tag{2.4}$$

Среднее значение класса $\mu_1(t)$ вычисляется как:

$$\mu_1(t) = [\sum_{i=0}^{t} ip(i)] / P_1,\ \mu_2(t) = [\sum_{i=t+1}^{L-1} ip(i)] / P_2 \quad (2.5)$$

Общий алгоритм бинаризации Otsu состоит из следующих этапов:

1) вычисление гистограммы исходного изображения;

2) для каждого возможного значения **t**:

2.1) рассчитать σ_b^2 **(t)**;

2.2) если σ_b^2 **(t)** больше текущего максимального значения, изменить максимальное значение на σ_b^2 **(t),** сохранить значение t.

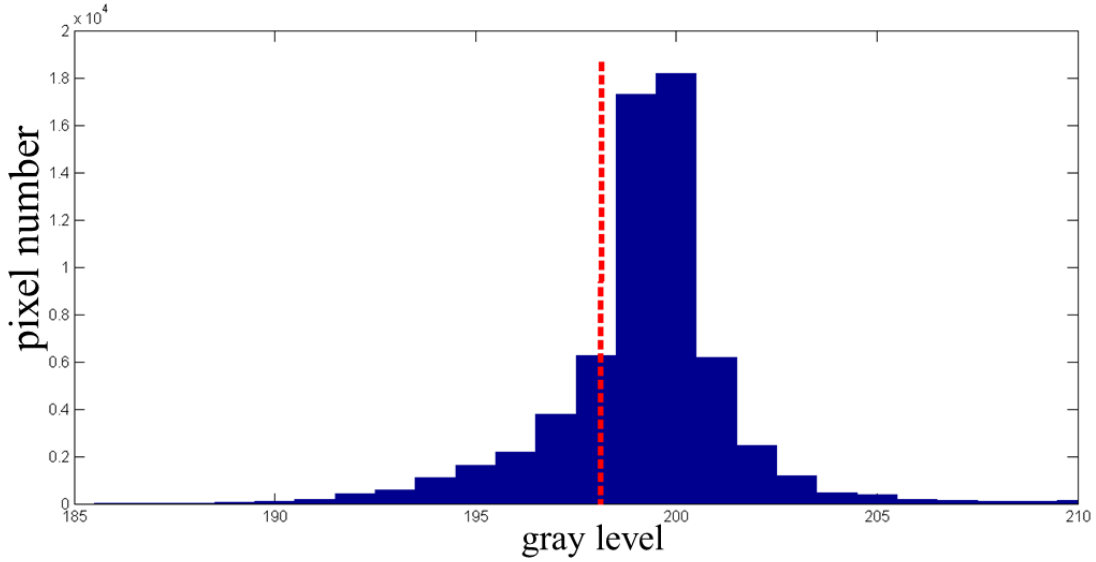

Рисунок 2.9

Гистограмма изображения со значением выбранного порога методом Отсу

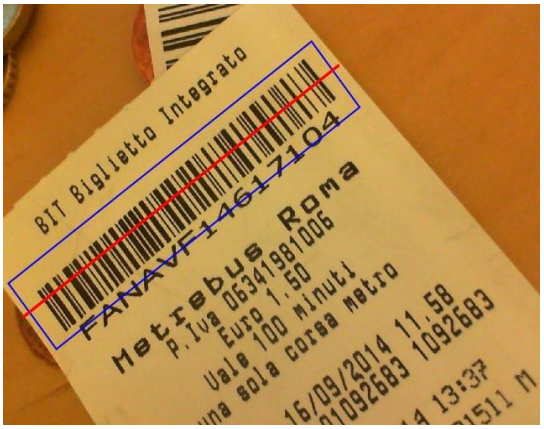

Рисунок 2.10

Перед созданием двоичного представления изображения

Рисунок 2.11
После создания двоичного представления изображения

Исходное цветное изображение преобразится в чёрно-белое, пройдя первый этап обработки.

2) Построить сетку изображения (20 x 15 ячеек). Двоичный образ в целом не предоставляет много информации о его содержимом. Чтобы понять структуру, содержащуюся в изображении, она должна быть разделена на более мелкие части, которые затем описываются независимо друг от друга. Полученное описание затем сравнивается с нашей гипотезой (линии, близкие друг к другу с аналогичной ориентацией) и либо принимаются во внимание для дальнейшей обработки, либо отбрасываются. Теперь каждая ячейка сетки должна быть оценена и классифицирована на основе свойств нашей гипотезы. В основном это означает, что каждая ячейка проверяется, чтобы определить, содержит ли она только параллельные линии. Если это так, она считается ценной и будет передана следующему шагу. Чтобы определить, содержит ли ячейка линии, которые, в лучшем случае, параллельны, бары должны быть нормализованы по ширине (1px ширина), затем разделены друг от друга, после чего следует оценка ориентации каждой линии. Затем каждая линия в ячейке сравнивается с остальными для определения параллельности. Наконец, если большинство линий параллельны друг другу, вычисляется средний угол. На рисунке 2.12 показана полученная после 2-го этапа сетка.

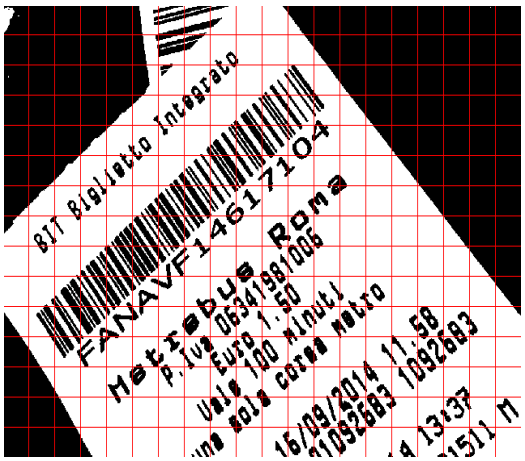

Рисунок 2.12
Разбиение изображения на сетку

3) *Извлечение скелета каждой ячейки*. Данный этап начинается с нормализации ширины до 1px, которая выполняется с помощью метода, известного как скелетизация. Основной целью любого алгоритма скелетизации является максимальное утоньшение всех линий на бинарном изображении, вследствие чего они получают толщину 1px. Скелетизация изображения, как правило, реализуется за несколько итераций. Если после полного прохода по изображению были удалены какие-либо пиксели, то необходимо повторить проход. Такое повторение должно быть до тех пор, пока во время прохода не будет удален ни один пиксель. Пример процесса скелетизации представлен на рисунке 2.13.

Рисунок 2.13
Итерационный процесс скелетизации

После прохождения данного этапа обработки получается «скелет» изображения, представленный на рисунке 2.14.

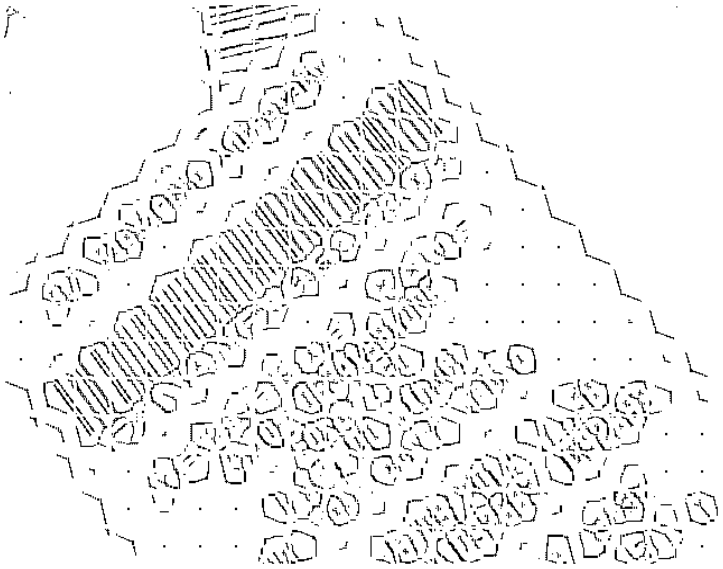

Рисунок 2.14
Извлечение скелета каждой ячейки

4) Маркировка компонент. Чтобы выяснить, содержит ли каждая ячейка параллельные линии, скелетированное содержимое, должно быть разделено на отдельные части (линии). Это может быть достигнуто путем применения метода, называемого меткой связных компонентов, который разделяет данную область на ее отдельные компоненты. В случае шаблона штрих-кода все линии в ячейке разделяются на отдельные объекты, а затем взвешиваются по их углу наклона. Каждая ячейка обрабатывается индивидуально, поэтому цвета между различными ячейками могут повторяться. Цвет каждого компонента в ячейке уникален и обозначает метку, заданную алгоритмом. На следующем рисунке 2.15 показаны расширенные версии двух извлеченных ячеек. Левая часть указывает на возможную область штрих-кода, тогда как правая не содержит ничего кроме шума. Каждая метка может затем рассматриваться как возможный штрих из штрих-кода. Чтобы иметь возможность классифицировать такое представление, необходимо вычеслить количественное значение для каждого бара, которое затем можно сравнить с другими компонентами.

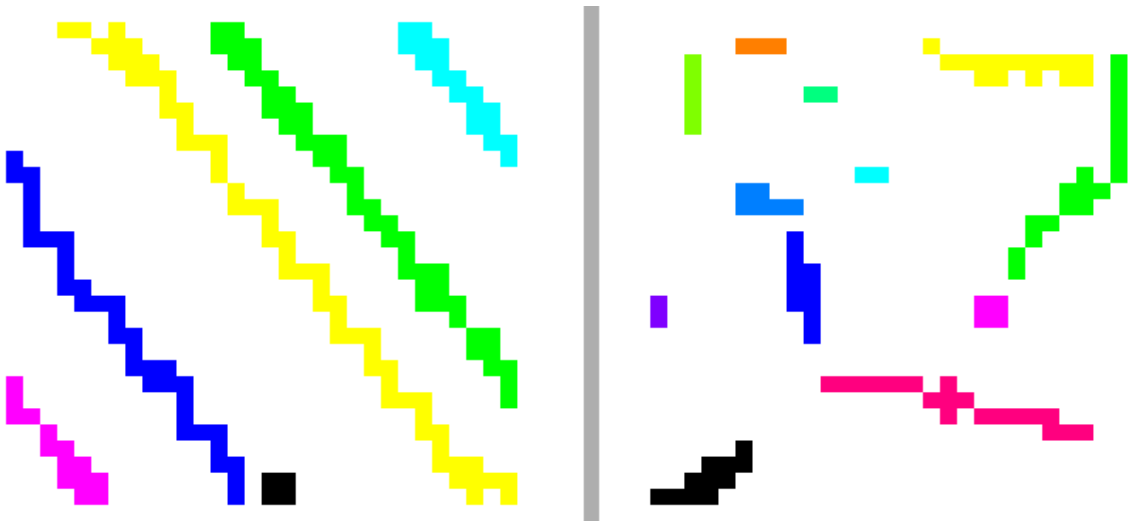

Рисунок 2.15
Маркировка компонент

5) Определение ориентации компонент. Ориентация отдельного компонента внутри ячейки вычисляется центральными моментами изображения. Этот метод обычно используется для извлечения информации, касающейся ориентации двоичного изображения. В этом случае двоичное изображение представлено помеченными компонентами. Каждый компонент можно рассматривать как свое собственное двоичное изображение, центральные моменты изображения которого можно вычислить. Как показано в приведённой ниже формуле (2.6), ориентация (обозначенная как θ) двоичного изображения может быть определена его центральными моментами (μ).

$$\Theta = \frac{1}{2}\arctan\left(\frac{2\mu'_{11}}{\mu'_{20} - \mu'_{02}}\right) \tag{2.6}$$

Центральные моменты (μ) вычисляются по формуле (2.7) с использованием начальных моментов (M) и центроида (x, y), которые должны быть заранее вычислены.

$$\mu'_{11} = M_{11}/M_{00} - \bar{x}\bar{y}$$
$$\mu'_{02} = M_{02}/M_{00} - \bar{y}^2 \quad (2.7)$$
$$\mu'_{20} = M_{20}/M_{00} - \bar{x}^2$$

Формула (2.8) вычисляет компоненты центроида (x, y), используя начальные моменты (M).

$$\bar{x} = M_{10}/M_{00}$$
$$\bar{y} = M_{01}/M_{00} \quad (2.8)$$

Так как нам нужны моменты изображения до второго порядка, формула (2.9) показывает вычисление каждого отдельного момента. Сумма по x и y обозначает итерацию по всему изображению, тогда как I (x, y) указывает значение пикселя в позиции x, y. В этом случае значение может быть либо 0, либо 1, поскольку мы работаем с бинарным изображением.

$$M_{00} = \sum_x \sum_y I(x,y)$$
$$M_{10} = \sum_x \sum_y xI(x,y)$$
$$M_{01} = \sum_x \sum_y yI(x,y)$$
$$M_{11} = \sum_x \sum_y xyI(x,y) \quad (2.9)$$
$$M_{20} = \sum_x \sum_y x^2I(x,y)$$
$$M_{02} = \sum_x \sum_y y^2I(x,y)$$

После этого шага каждому компоненту присваивается свойство со значением определенного угла.

6) Определение качества ячейки. Теперь, когда каждая ячейка содержит компоненты с их соответствующей ориентацией, мы можем начать классифицировать ячейки на основе этих свойств. Прежде всего, ячейки, содержащие 0 или только 1 компонент, немедленно отбрасываются и не

используются для дальнейшей обработки. Кроме того, компоненты, которые не покрывают по крайней мере 6px, исключаются из любых последующих вычислений, касающихся затронутой ячейки. Этот предварительно отфильтрованный список служит основой для определения однородности углов компонента по целой ячейке. В случае штрих-кода каждый компонент в ячейке должен быть параллелен друг другу. Однако из-за шума, искажений и других влияний это может быть не всегда так. Для определения сходства компонентов, содержащих шаблон штрих-кода, применяется простая методика кластеризации. Прежде всего, все углы группируются с определенным порогом, после чего выбирается кластер с самым большим количеством членов. Если размер члена этого кластера больше 3/4-го от исходного набора, ячейка окончательно считается частью шаблона штрих-кода. С этого момента эта ячейка называется патчем, который содержит следующую информацию:

- Индекс (уникальный идентификатор внутри сетки);
- Ограничивающий прямоугольник, определяющий ячейку;
- Все компоненты и связанные с ними моменты;
- Средний угол (вычисленный из кластера);
- Вектор, указывающий направление ориентации.

На следующем рисунке 2.13 выделены участки (ячейки), которые были классифицированы как штрих-код.

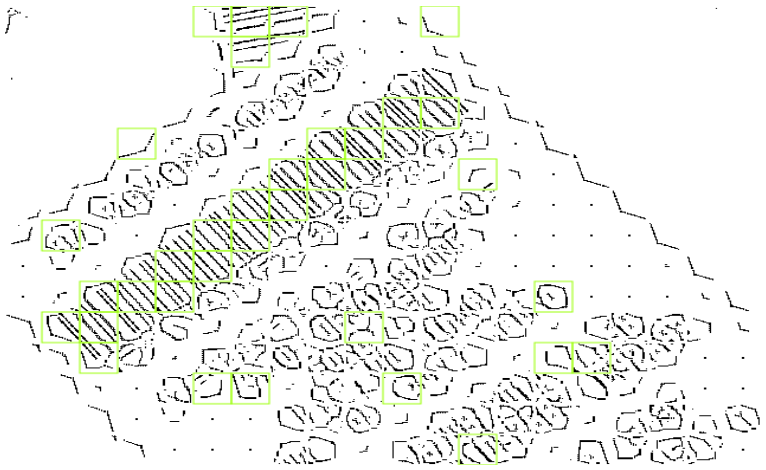

Рисунок 2.16
Определение качества ячейки

На рисунке 2.16 можно видеть, что некоторые ячейки ошибочно классифицировались как часть штрих-кода. Эти ложные срабатывания можно обнаружить, просто найдя прилегающие области (состоящие из более чем одной ячейки) и отбросив все остальные.

7) Поиск соседних ячеек. Ячейки можно считать частью штрих-кода, если они являются соседями и имеют общие свойства. Эта группировка выполняется простым алгоритмом рекурсивного компонента-маркировки, который работает с использованием подобия ориентации. Чтобы патч (компонент) считался элементом штрих-кода, его вектор должен быть сонаправлен с вектором его соседа. Для учета отклонений, вызванных искажением и другими геометрическими влияниями, ориентация может отклоняться до 5%. Изображение, показанное ниже, иллюстрирует связанные ячейки, где цвет определяет связь с определенной группой. Единственная группа, содержащая более одного патча, - желтая. Иногда даже соседние ячейки окрашены по-разному. Это происходит, если ориентация этих патчей сильно отличается (> 5%). Результат представлен на рисунке 2.17.

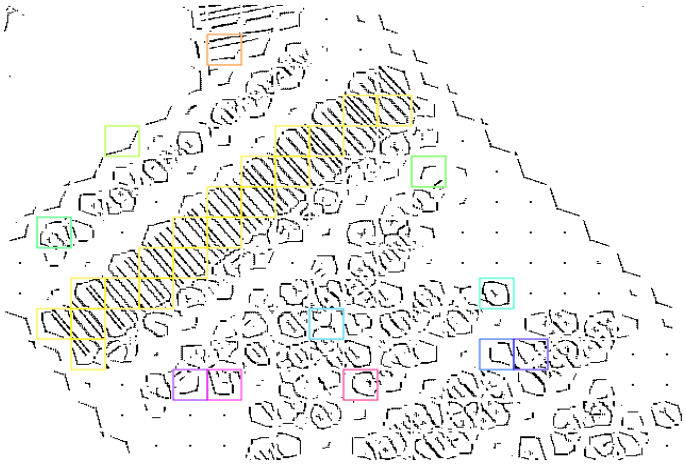

Рисунок 2.17
Поиск соседних ячеек

8) Выбор групп ячеек. Далее выбираются те группы, которые, скорее всего, содержат штрих-код. Поскольку в изображении могут одновременно присутствовать несколько штрих-кодов, группы сначала сортируются, а затем фильтруются по их числу элементов (патчей). Промежуточный результат представлен на рисунке 2.18.

Рисунок 2.18
Выбор групп ячеек

В примере, показанном выше, остается только желтая группа, которая и передается на следующий шаг - создание ограничивающего прямоугольника.

9) Создание ограничительной рамки. Наконец, доступна вся информация, необходимая для определения местоположения штрих-кода. На этом последнем

шаге создается минимальная ограничивающая рамка, которая охватывает все патчи в одной группе. Сначала вычисляется средний угол содержащихся компонентов, который затем используется для поворота рамки. После этого находится сама ограничивающая рамка, используя координаты самых крайних элементов и затем поворачивается на вычисленный угол (рисунок 2.19).

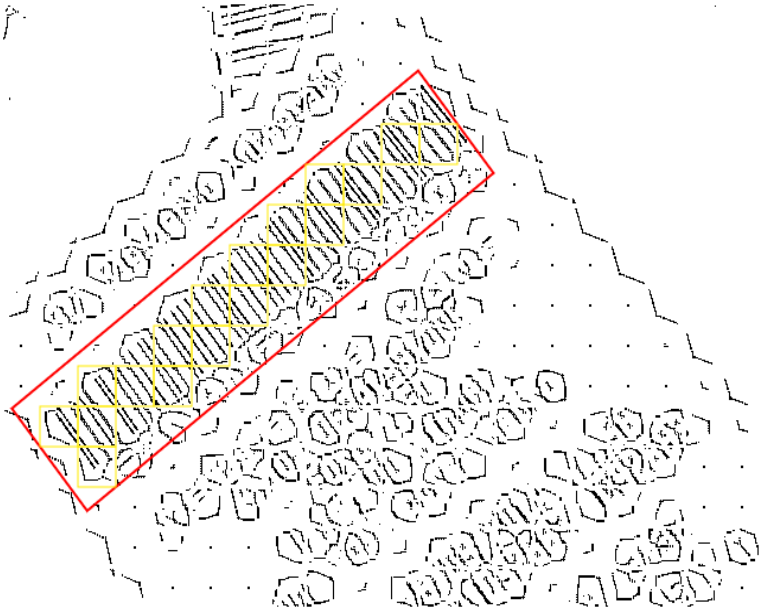

Рисунок 2.19
Создание ограничительной рамки

2.4 Использование Python и OpenCV

Рассмотрим решение задачи с примерами кода для нахождения ШК. Целевой объект представлен на рисунке 2.20

Рисунок 2.20
Исходное изображение

Прежде всего нужно сделать импорт необходимых пакетов. Для решения задачи потребуются **NumPy** для работы с числами, **agparse** для парсинга аргументов командной строки и **cv2** для связи с **OpenCV**.

```
import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True, help = "path to the image file")
args = vars(ap.parse_args())
```

Далее загружается изображение image, после чего происходит преобразование его цветового режима в оттенки серого.

```
image = cv2.imread(args["image"])
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

Затем, нужно использовать оператор Собеля (с выставленным ksize = -1), чтобы вычислить величину градиента серой картинки в вертикальном и горизонтальном направлениях:

```
gradX = cv2.Sobel(gray, ddepth = cv2.cv.CV_32F, dx = 1, dy = 0, ksize = -1)
gradY = cv2.Sobel(gray, ddepth = cv2.cv.CV_32F, dx = 0, dy = 1, ksize = -1)
```

После этого вычитается y-градиент из x-градиента:

```
gradient = cv2.subtract(gradX, gradY)
gradient = cv2.convertScaleAbs(gradient)
```

В итоге получается изображение с высоким значением горизонтального градиента и низким значением вертикального (рисунок 2.21):

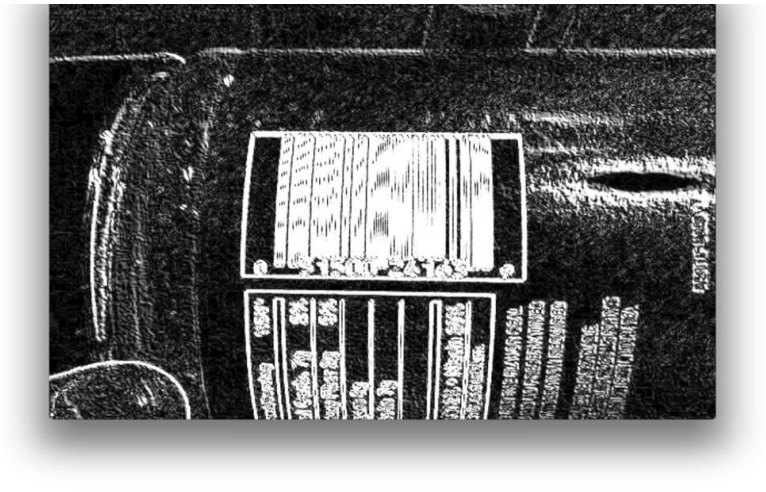

Рисунок 2.21
изображение с высоким значением горизонтального градиента и низким значением вертикального

Следующий шаг — устранить шум на изображении и сфокусироваться сугубо на области со штрих-кодом. Для этого используется average blur с ядром размера 9x9. Это поможет сгладить высокочастотный шум на картинке с градинентами. Затем проводится бинаризация размытого изображения. Каждый пиксель со значением не выше 225 превращается в 0 (чёрный), а остальные — в 255 (белый).

```
blurred = cv2.blur(gradient, (9, 9))
(_, thresh) = cv2.threshold(blurred, 225, 255, cv2.THRESH_BINARY)
```

Результат представлен на рисунке 2.22:

Рисунок 2.22
Применение ядра для сглаживания высокочастотного шума и бинаризации

На изображении выше, между вертикальными полосками штрих-кода есть пространство. Чтобы его закрыть и облегчить алгоритму определение области ШК, нужно произвести ряд простых морфологических операций:

```
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (21, 7))
closed = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)
```

Сначала создаётся прямоугольник с помощью cv2.getStructuringElement. Ширина ядра больше его высоты, что позволяет перекрыть пространство между вертикальными полосками штрихкода. Далее применяется ядро к бинаризированному изображению, замазывая пространство между полосками. Вследствие чего пробелы почти полностью закрылись (рисунок 2.23):

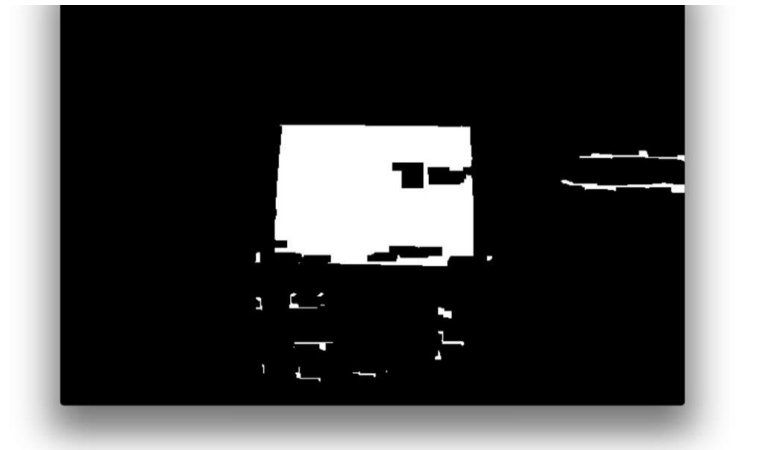

Рисунок 2.23
Пробелы почти полностию закрылись

На рисунке 2.17 остались некоторые светлые пятна, которые не имеют отношения к штрих-коду и способны помешать точно определить его контур. Для избавления от этих пятен производится четыре итерации эрозии, за которыми следуют четыре итерации дилатация.

```
closed = cv2.erode(closed, None, iterations = 4)
closed = cv2.dilate(closed, None, iterations = 4)
```

Эрозия уберёт белые пиксели с изображения, удаляя мелкие пятна, а дилатация не позволит крупным белым областям уменьшиться. После серии эрозий и дилатаций на рисунке 2.24 можно увидеть, что ненужные объекты были удалены и осталась только область штрих-кода:

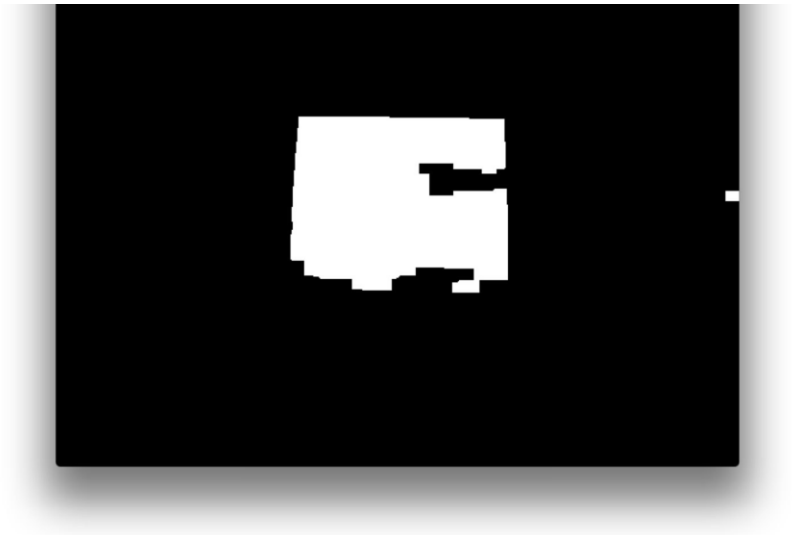

Рисунок 2.24
Ненужные объекты были удалены после серии эрозий и дилатаций

Далее ищется ограничивающая рамка для области со штрих-кодом. Самый большой контур на изображении можно найти с помощью cv2.findContours, который (если обработка была произведена корректно) точно соотносится с целевой областью:

```
(cnts, _) = cv2.findContours(closed.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
c = sorted(cnts, key = cv2.contourArea, reverse = True)[0]
```

Затем определяется минимальный ограничивающий прямоугольник, в который заключается этот самый большой контур:

```
rect = cv2.minAreaRect(c)
box = np.int0(cv2.cv.BoxPoints(rect))
```

после чего наконец отображаем найденный штрихкод (см. рисунок 2.25):

```
cv2.drawContours(image, [box], -1, (0, 255, 0), 3)
cv2.imshow("Image", image)
cv2.waitKey(0)
```

Рисунок 2.25
Выделенная облать со штрих-кодом

В итоге, в основе реализованного алгоритма лежат следующие действия:

1) Вычисление размера градиента по осям x и y;
2) Отделение вертикального градиента от горизонтального, чтобы выявить область штрих-кода;

3) Применение размытия и бинаризации;
4) Применение ядра к бинаризированной картинке для удаления «пробелов» между полосками;
5) Произведение серии эрозий и дилатаций;
6) Нахождение на изображении самого большого контура, который и будет являться областью штрих-кода.

На рисунке 2.26 можно увидеть UML-диаграмму работы решения с использованием JavaScript и QuaggaJS, а на рисунке 2.27 – работу решения с использованием Python и OpenCV.

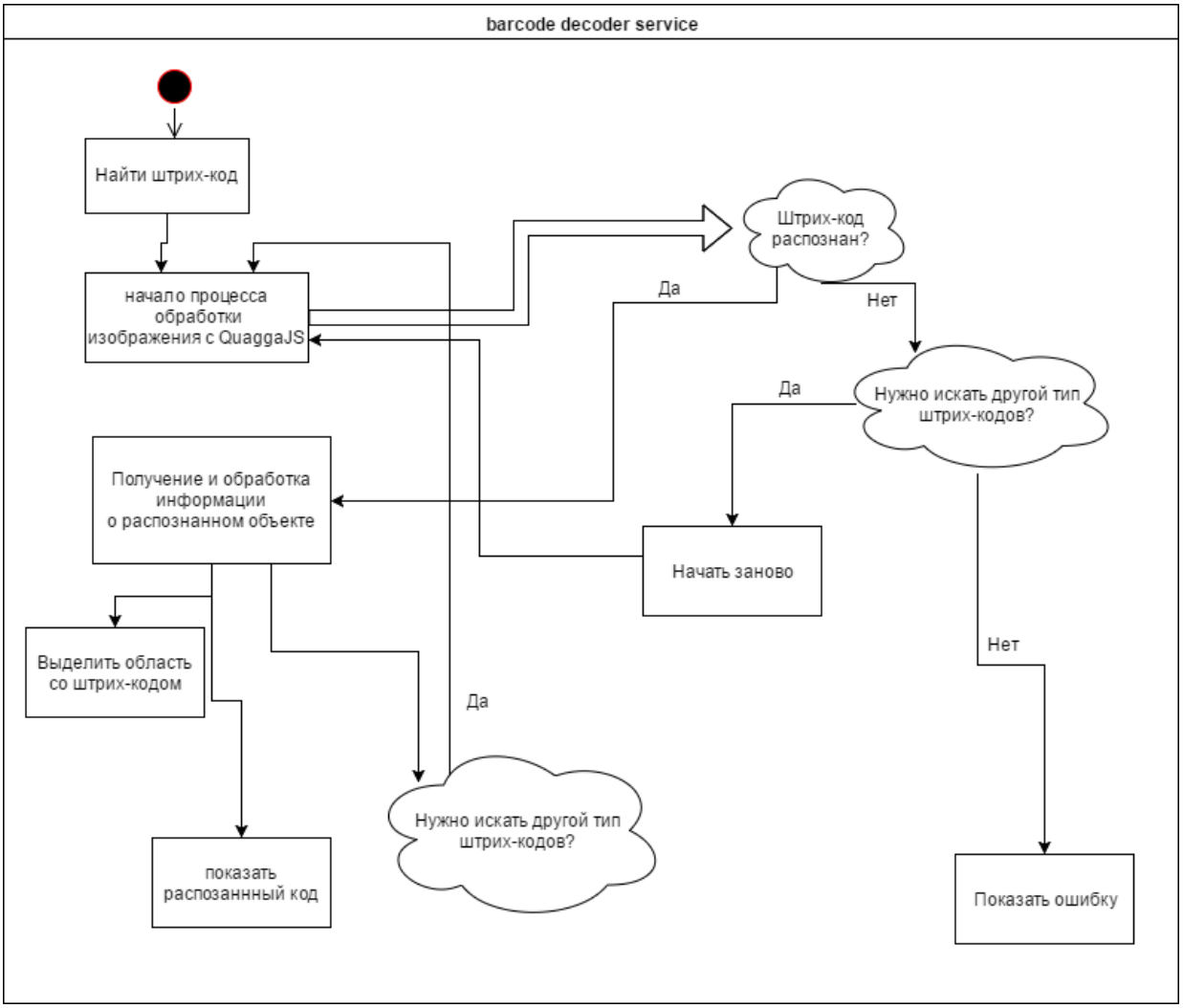

Рисунок 2.26
UML-диаграмма работы JavaScript и QuaggaJS

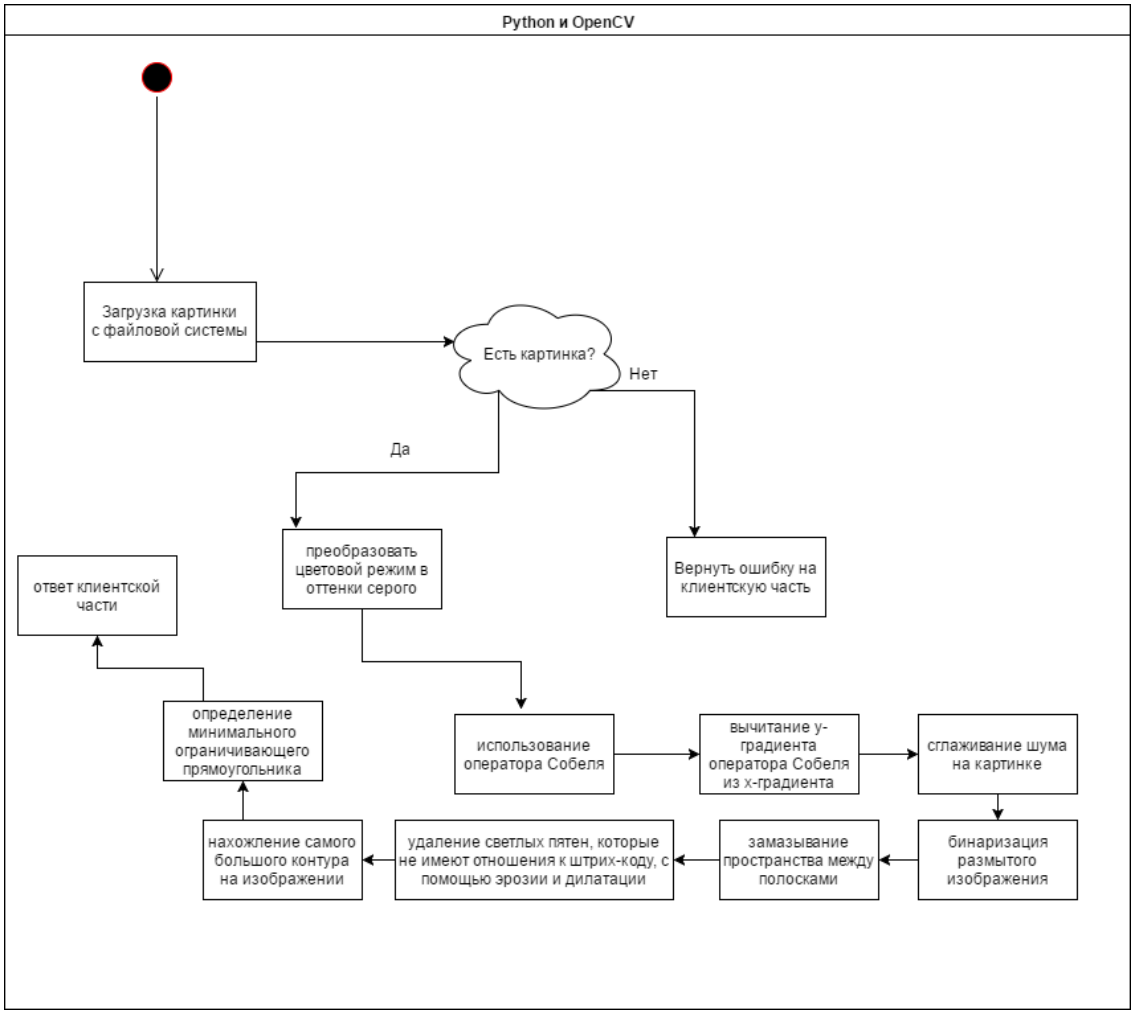

Рисунок 2.27
UML-диаграмма работы Python и OpenCV

Вывод второй главы можно сделать следующий: засчёт того, что алгоритм на python не учитывает ориентацию штрих-кода, а также не всегда корректно обрабатывает изображения, на которых присутствует много шумов, скорость его работы оказалась выше скорости алгоритма, реализованного в QuaggaJS. Однако, большое количество настроек этой библиотеки, позволяют вручную оптимизировать алгоритм, основываясь на предположениях о размещении ШК на картинке, его вероятностных размерах, ориентации. Это даёт возможность задать подходящие под определённые нужды параметры, что может значительно снизить нагрузку на процессор устройства, выполняющего работу по обнаружению и распознаванию объекта. Обработка изображений с помощью разных алгоритмов на

разных языках программирования, с использованием различных библиотек и встроенных функций, была выполнена за неодинаковое время время. Результаты тестирования описаны в пункте 2.5

2.5 Результаты тестирования

Все тесты были проведены на машине, которая имеет конфигурацию, указанную в табдице 2.1.

Операционная система	Windows 10 Pro
Процессор	AMD A6-3400M APU with Radeon(tm) HD Graphics 1.40 GHz
Оперативная память	4 GB
Тип архитектуры системы	64-битная архитектура на базе процессора x64

Таблица 2.1
Конфигурация машины, на которой проводились эксперименты

Результаты тестирования работы функционала отображены в таблице 2.2. Тесты проводились на изображениях, указанных в таблице 2.3. По проведённым тестам можно сделать вывод, что использование библиотеки QuaggaJS даёт более точные результаты и справляется с неверной ориентацией изображения хотя тратит немного больше времени на поиск и распознавание штрих-кода засчёт проверок на наличие нескольких стандартных типов ШК. Есть возможность подобрать настройки в конфигурационном объекте таким образом, чтобы работа смогла ускориться, а результат был более стабильным. Работу алгоритма, написанного на python с использованием OpenCV, можно улучшить путём применения дополнительных фильтров, встроенных в функционал библиотеки, рассмотрением проблемы ориентации картинки, а также можно увеличить количество итераций эрозии и дилатации, что заметно скажется на времени работы решения.

	№1	№2	№3	№4	№5	№6
Python & OpenCV	1.347s	1.53s	1.785s	Распознано неверно	Не распознано	1.56s
QuaggaJS	2.54s	1.921s	2.421s	2.24s	2.781s	2.17s

Таблица 2.2
Резальтаты тестирования

Верхние ячейки с номерами в таблице выше указывают номер тестовой картинки из таблицы 2.3

Рисунок 3.1
Тестовое изображение №1

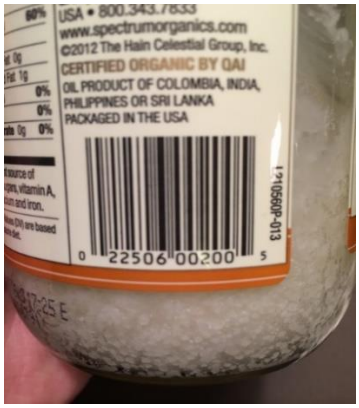

Рисунок 3.2
Тестовое изображение №2

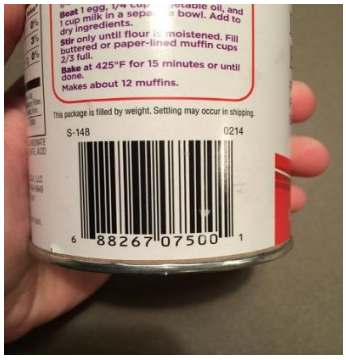

Рисунок 3.3
Тестовое изображение №3

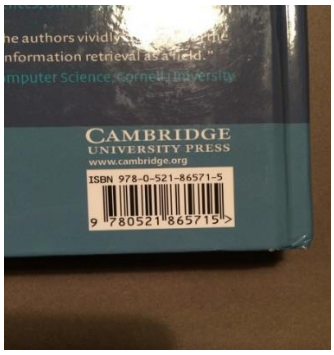

Рисунок 3.4
Тестовое изображение №4

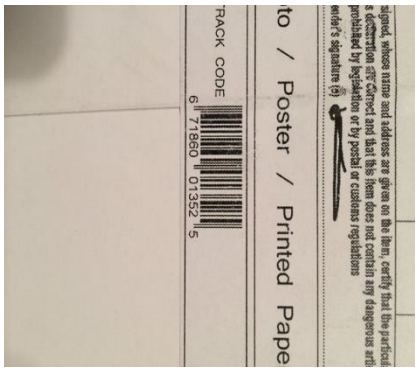

Рисунок 3.5
Тестовое изображение №5

Рисунок 3.6
Тестовое изображение №6

Таблица 2.3
Тестовые изображения

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломной работы были изучены алгоритмы компьютерного зрения, изучена библиотека QuaggaJS для реализации функционала с использованием языка JavaScript, а также язык python и применяемая вместе с ним библиотека OpenCV для обработки изображения, framework node.js, angular 2.

Было реализовано веб-приложение с функцией выбора метода распознавания штрих-кода по входному изображению, дальнейшей его обработкой и выделением найденной области цветным контуром.

Были проанализированы скорости работы алгоритмов, реализованных с использованием языков программирования JavaScript и Python.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Л. Шапиро, Дж. Стокман. Компьютерное зрение = Computer Vision. — М.: Бином. Лаборатория знаний, 2006. — 752 с.
2. Дэвид Форсайт, Жан Понс. Компьютерное зрение. Современный подход = Computer Vision: A Modern Approach. — М.: «Вильямс», 2004. — 928 с.
3. А.А. Лукьяница, А.Г. Шишкин. Цифровая обработка видеоизображений. — М.: «Ай-Эс-Эс Пресс», 2009. — 518 с.
4. Желтов С.Ю. и др. Обработка и анализ изображений в задачах машинного зрения. — М.: Физматкнига, 2010. — 672 с.
5. Р. Сцелиски. Компьютерное зрение: алгоритмы и приложения = Computer Vision: Algorithms and Applications. — М.: Спрингер, 2010. — 979 с.
6. QuaggaJS Documentation [Электронный ресурс] – Режим доступа: https://serratus.github.io/quaggaJS/ - Дата доступа: 20.04.2017.
7. How barcode-localization works in QuaggaJS [Электронный ресурс] – Режим доступа: http://www.oberhofer.co/how-barcode-localization-works-in-quaggajs/ - Дата доступа: 25.04.2017
8. Анализ алгоритмов компьютерного зрения [Электронный ресурс] – Режим доступа: http://www.arealidea.ru/articles/analiz-algoritmov-kompyuternogo-zreniya-poiska-obektov-i-sravneniya-izobrazheniy/ - Дата доступа: 10.05.2017.

ПРИЛОЖЕНИЕ А.

Код программы

Главный модуль программы, который импортирует все важные зависимости и загружает приложение в окне браузера находится в файле **app.module.ts** и выглядит следующим образом:

```
import 'hammerjs';

import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';
import { BrowserAnimationsModule } from '@angular/platform-
browser/animations';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { GetStartedComponent } from './get-started/get-
started.component';
import { HttpModule } from '@angular/http';
import { MaterialModule } from '@angular/material';
import { NgModule } from '@angular/core';
import { PostsComponent } from './posts/posts.component';
import { PostsService } from './posts.service';
import { StartPageComponent } from './start-page/start-page.component';
import { PythonPageComponent } from './python-page/python-
page.component';
import { JsPageComponent } from './js-page/js-page.component';
import { BarcodeComponent } from './js-page/barcode/barcode.component';
import { MediaStreamComponent } from './js-page/barcode/media-
stream/media-stream.component';
import { InputFieldComponent } from './js-page/barcode/input-
field/input-field.component';
import { InstantSearchComponent } from './js-page/barcode/instant-
search/instant-search.component';
import { BarcodeValidatorService } from './js-
page/barcode/services/barcode-validator.service';
import { BarcodeDecoderService } from './js-
page/barcode/services/barcode-decoder.service';
import { FlexLayoutModule } from '@angular/flex-layout';
import { PythonBarcodeService } from './python-page/python-
barcode.service';
import { AboutComponent } from './about/about.component';
```

```
import { ContactsComponent } from './contacts/contacts.component';

@NgModule({
  declarations: [
    AppComponent,
    StartPageComponent,
    PostsComponent,
    GetStartedComponent,
    PythonPageComponent,
    JsPageComponent,
    BarcodeComponent,
    MediaStreamComponent,
    InputFieldComponent,
    InstantSearchComponent,
    AboutComponent,
    ContactsComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule,
    HttpModule,
    AppRoutingModule,
    MaterialModule,
    FlexLayoutModule
  ],
  providers: [BarcodeValidatorService, BarcodeDecoderService,
PostsService, PythonBarcodeService],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

Код программы, который использует библиотеку QuaggaJS, когда пользователь выбрал опцию распознавания штрих-кода в режиме реального времени с помощью камеры устройства:

```
import { Component, OnInit, OnDestroy, ViewChild, AfterContentInit }
from '@angular/core';
import { BarcodeDecoderService } from "../services/barcode-
decoder.service";
import { BarcodeValidatorService } from "../services/barcode-
validator.service";
```

```
import { Subject } from "rxjs/Subject";


@Component({
  selector: 'app-media-stream',
  templateUrl: './media-stream.component.html',
  styleUrls: ['./media-stream.component.scss']
})
export class MediaStreamComponent implements OnInit, OnDestroy,
AfterContentInit {

  lastResult: any;
  message: any;
  error: any;

  code$ = new Subject<any>();

  @ViewChild('interactive') interactive;

  constructor(private decoderService: BarcodeDecoderService, private
barcodeValidator: BarcodeValidatorService) {};

  ngOnInit() {

    this.decoderService.onLiveStreamInit();
    this.decoderService.onDecodeProcessed();

    this.decoderService.onDecodeDetected()
        .then(code => {
          this.lastResult = code;
          this.decoderService.onPlaySound();
          this.code$.next(code);
        })
        .catch((err) => this.error = `Something Wrong: ${err}`);

    this.barcodeValidator.doSearchbyCode(this.code$)
        .subscribe(
          res => this.message = res,
          err => {
            this.message = `An Error! ${err.json().error}`
          }
        );
  }

  ngAfterContentInit() {
    this.interactive.nativeElement.children[0].style.position =
'absolute';
```

```
  }

  ngOnDestroy() {
    this.decoderService.onDecodeStop();
  }

}
```

Код программы, который использует библиотеку QuaggaJS, когда пользователь выбрал опцию распознавания штрих-кода с функцией загрузки исходного изображения с файловой системы устройства:

```
import { Component, ViewChild, OnDestroy, OnInit } from
'@angular/core';
import { DomSanitizer } from "@angular/platform-browser";
import { BarcodeValidatorService } from "../services/barcode-
validator.service";
import { BarcodeDecoderService } from "../services/barcode-
decoder.service";
import { Subject } from "rxjs/Subject";
import 'rxjs/add/operator/map';

@Component({
  selector: 'app-input-field',
  templateUrl: './input-field.component.html',
  styleUrls: ['./input-field.component.scss']
})
export class InputFieldComponent implements OnInit, OnDestroy {

  @ViewChild('isbn') isbn;
  @ViewChild('fileInputbox') fileInputbox;

  resultUrl: any;
  resultCode: any;
  startProgress: boolean = false;
  error: any;
  message: string;

  code$ = new Subject<any>();


  constructor(private sanitizer: DomSanitizer,
              private barcodeValidator: BarcodeValidatorService,
              private decoderService: BarcodeDecoderService) {}
```

```
  ngOnInit() {
  this.decoderService.onDecodeProcessed();
  this.decoderService.onDecodeDetected()
      .then(code => {
        debugger;
      })
      .catch((err) => {
        debugger;
      });

  this.barcodeValidator.doSearchbyCode(this.code$)
      .subscribe(
        res => this.message = res,
        err => {
          this.message = `An Error! ${err.json().error}`
        }
      );
}

sanitize(url: string) {
  return this.sanitizer.bypassSecurityTrustUrl(url);
}

setStartProgress() {
  this.startProgress = !this.startProgress;
}

onChange(e) {
  const file = URL.createObjectURL(e.target.files[0]);
  this.decoderService.onDecodeSingle(file)
      .then(code => {
        this.resultUrl = this.sanitize(file);
        this.isbn.value = code;
        this.resultCode = code;
        this.decoderService.onPlaySound();
        this.code$.next(code);
        this.setStartProgress();
      })
      .catch(e => {
        this.resultUrl = '';
        this.resultCode = '';
        this.isbn.value = '';
        this.setStartProgress();
        this.error = `Something is wrong: ${e}`;
      });
}
```

```
  onCancel(e) {
    this.setStartProgress();
    this.error = `Something is wrong: Please Select An Image`;
  }

  onClick() {
    this.setStartProgress();
    this.fileInputbox.nativeElement.click();
    this.error = null;
  }

  ngOnDestroy() {
    console.info('Stopped!')
  }
}
```

Основная логика распознавания ШК с помощью QuaggaJS находится в файле **barcode-decoder.service.ts**. Данный сервис предоставляет публичные и приватные методы, которые используют модули приложения по распознаванию ШК в режиме реального времени и через файловую систему соответственно. Ниже приведён его код:

```
import { Injectable } from '@angular/core';
import { DECODER_CONFIG, DECODER_LIVE_CONFIG } from
"../../../config/decoder-config";
import Quagga from 'Quagga';

@Injectable()
export class BarcodeDecoderService {

  sound = new Audio('assets/audio/barcode.wav');

  constructor() {}

  onDecodeSingle(src) {
    DECODER_CONFIG.src = src;

    return new Promise((resolve, reject) => {
      Quagga.decodeSingle(DECODER_CONFIG, result => {
        if (!result || typeof result.codeResult === 'undefined') {
          reject('File Cannot be Decode, Please Try a Valid Barcode;');
        }
```

```
        resolve(result.codeResult.code);
      });
    });
  }

  private setLiveStreamConfig() {
    DECODER_LIVE_CONFIG.inputStream = {
      type: "LiveStream",
      constraints: {
        width: {min: 533},
        height: {min: 400},
        facingMode: "environment",
        aspectRatio: {
          min: 1,
          max: 2
        }
      }
    };
    return DECODER_LIVE_CONFIG;
  }

  onLiveStreamInit() {
    const state = this.setLiveStreamConfig();
    Quagga.init(state, (err) => {
      if (err) {
        return console.error(err);
      }
      Quagga.start();
    });
  }

  onProcessed(result: any) {
    let drawingCtx = Quagga.canvas.ctx.overlay,
      drawingCanvas = Quagga.canvas.dom.overlay;

    if (result) {
      if (result.boxes) {
        drawingCtx.clearRect(0, 0,
parseInt(drawingCanvas.getAttribute("width")),
parseInt(drawingCanvas.getAttribute("height")));
        result.boxes.filter(function(box) {
          return box !== result.box;
        }).forEach(function(box) {
          Quagga.ImageDebug.drawPath(box, {
            x: 0,
            y: 1
          }, drawingCtx, {
```

```
            color: "green",
            lineWidth: 2
          });
        });
      }

      if (result.box) {
        Quagga.ImageDebug.drawPath(result.box, {
          x: 0,
          y: 1
        }, drawingCtx, {
          color: "#00F",
          lineWidth: 2
        });
      }

      if (result.codeResult && result.codeResult.code) {
        Quagga.ImageDebug.drawPath(result.line, {
          x: 'x',
          y: 'y'
        }, drawingCtx, {
          color: 'red',
          lineWidth: 3
        });
      }

    }
  }

  onDecodeProcessed() {
    Quagga.onProcessed(this.onProcessed);
  }

  onDecodeDetected() {
    return new Promise((resolve, reject) => {
      Quagga.onDetected(result => {
        if (!result || typeof result.codeResult === 'undefined') {
          reject('Cannot be Detected, Please Try again!');
        }
        resolve(result.codeResult.code);
      });
    });
  }

  onDecodeStop() {
    Quagga.stop();
    console.info('Camera Stopped Working!');
```

```
  }

  onPlaySound() {
    this.sound.play();
  }
}
```

Далее представлен сервис, который получает информацию на основе переданного ему распознанного кода на штрих-коде и делает запрос на общедоступный API. После этого, в случае если ШК был правильно распознан и такой код существует, то сервер присылает подробную информацию о целевом продукте, такую как страна производитель, город, индекс, дата изготовления и др.:

```
import { Injectable } from '@angular/core';
import { Http } from "@angular/http";
import { Observable } from "rxjs/observable";
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
import 'rxjs/add/operator/switchMap';

@Injectable()
export class BarcodeValidatorService {

  constructor(private _http: Http) { }

  private endpoints = {
    search: 'https://mutec.gomus.de/api/v3/barcodes/'
  };

  doSearchbyCode(codes: Observable<any>, debounceMs = 400) {
    return codes
      .debounceTime(debounceMs)
      .distinctUntilChanged()
      .switchMap(code => this.rawSearchByCode(code));
  }

  rawSearchByCode(code): Observable<any> {
    return this._http.get(`${this.endpoints.search}${code}`)
               .map(response => response.json())
               .catch(this.handleError);
  }
```

```
  private handleError(error: any): Promise<any> {
    return Promise.reject(error.message || error);
  }

}
```

В файле **decoder-config.ts** находится конфигурационный объект, который импортируется и передаётся в инициализирующую QuaggaJS функцию. В нём происходит указание всех настроек для работы библиотеки. Выглядит он следующим образом:

```
export let DECODER_CONFIG;

DECODER_CONFIG = {
  inputStream: {
    size: 600
  },
  locator: {
    patchSize: "medium",
    halfSample: false
  },
  numOfWorkers: 1,
  decoder: {
    readers: ['ean_reader', 'code_128_reader', 'ean_8_reader',
'code_39_reader', 'code_39_vin_reader',
      'codabar_reader', 'upc_reader', 'upc_e_reader', 'i2of5_reader']
  },
  locate: true,
  src: null
};

export let DECODER_LIVE_CONFIG;
DECODER_LIVE_CONFIG = {
  locator: {
    patchSize: "medium",
    halfSample: false
  },
  numOfWorkers: 1,
  decoder: {
    readers: ['ean_reader', 'code_128_reader', 'ean_8_reader',
'code_39_reader', 'code_39_vin_reader',
      'codabar_reader', 'upc_reader', 'upc_e_reader', 'i2of5_reader']
  },
  locate: true,
```

```
};
```

Если пользователь выберет табличку с опцией распознавания штрих-кода с помощью Python & OpenCV, то запущенный сервер приложения получит запрос и вызовет соответствующий python-скрипт. Код сервера, который слушает данный порт, находится в файле **api.js** и выглядит следующим образом:

```
const express = require('express');
const router = express.Router();

const axios = require('axios');
const API = 'https://jsonplaceholder.typicode.com';

const myPythonScriptPath = '../detect_barcode.py';
const PythonShell = require('python-shell');

router.get('/', (req, res) => {
  res.send('api works');
});

router.get('/posts', (req, res) => {
  axios.get(`${API}/posts`)
    .then(posts => {
      res.status(200).json(posts.data);
    })
    .catch(error => {
      res.status(500).send(error)
    });
});

router.get('/barcode-python', (req, res) => {
  let options = {
    scriptPath: '../',
    args: ['--image=imagePath']
  };

  let pyshell = new PythonShell(myPythonScriptPath, options);

  pyshell.on('message', function (message) {
      console.log(message);
      res.json()
  });
```

```
  pyshell.end(function (err) {
      if (err){
          throw err;
      };

      console.log('finished');
  });

  res.send("Ok");
});

module.exports = router;
```

А сам python-скрипт с реализованным алгоритмом распознавания, который получает на входе исходное изображение и в конце всех операций показывает окно с данной картинкой и выделенной на ней областью со штрих-кодом, находится в файле **detect_barcode.py**. Ниже представлен его код:

```
import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True, help = "path to the
image file")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

gradX = cv2.Sobel(gray, ddepth = cv2.CV_32F, dx = 1, dy = 0, ksize = -
1)
gradY = cv2.Sobel(gray, ddepth = cv2.CV_32F, dx = 0, dy = 1, ksize = -
1)

gradient = cv2.subtract(gradX, gradY)
gradient = cv2.convertScaleAbs(gradient)

blurred = cv2.blur(gradient, (9, 9))
(_, thresh) = cv2.threshold(blurred, 225, 255, cv2.THRESH_BINARY)

kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (21, 7))
```

```
closed = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)

closed = cv2.erode(closed, None, iterations = 4)
closed = cv2.dilate(closed, None, iterations = 4)

(_, cnts, _) = cv2.findContours(closed.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
c = sorted(cnts, key = cv2.contourArea, reverse = True)[0]

rect = cv2.minAreaRect(c)
box = np.int0(cv2.boxPoints(rect))

cv2.drawContours(image, [box], -1, (0, 255, 0), 3)
cv2.imshow("Image", image)
print(box);
cv2.waitKey(0)
```