Содержание

Аннотация **4**

1 Введение **5**

1.1 Цель 5

1.2 Задачи 5

2 Создание Android приложения. **6**

2.1 Теория 6

2.2 Практика 7

3 Разработка серверного программного обеспечения. **10**

3.1 Теория 11

3.2 Практика 12

4 Получение изображения выравненной бумаги из исходного изображения, сфотографированного под углом **12**

4.1 Основные определения 13

4.2 Обнаружение листа бумаги 14

4.3 Преобразование фотографии, снятой под разными углами, для получения выровненного изображения бумаги 16

4.4 Реализация 16

5 Предобработка изображения, удаление теней и шумов. Подготовка изображения к детекции участков с текстом **17**

5.1 Алгоритм обработки 17

5.2 Реализация 18

6 Разбиение текста на изображении на отдельные строки для повышения точности определения порядка анимации. **19**

6.1 Теория 19

6.2 Практика 19

7 Разбиение строк на множество отдельных штрихов. **21**

7.1 Теория 22

7.2 Практика 23

8 Сортировка штрихов в порядке их написания внутри одной строки и сортировка строк на итоговом изображении. 24

8.1 Теория . . . 24

9 Отображение полученных пикселей в правильном порядке в файл, создание анимационного файла, передача файла обратно на основной код сервера. 25

9.1 Терминология . . . 25

9.2 Теория . . . 25

9.3 Практика . . . 29

10 Заключение 31

Аннотация

В данном курсовом проекте описывается разработка Android-приложения, которое предоставляет пользователю возможность сделать фотографию рукописного текста, по которой будет сгенерирован файл, содержащий анимацию, имитирующую написание текста с фотографии "от руки". Полученный файл может быть отправлен другим пользователям через сторонние сервисы. Кроме того, с помощью приложения можно открыть и воспроизвести анимационный файл, полученный от других пользователей.

Ключевые слова

Компьютерное зрение, android приложение, кодирование файла, обнаружение текста, вычисления на сервере, Python3, GOlang, Java, OpenCV

1 Введение

Раньше люди часто общались с помощью рукописных писем, сейчас же общение перешло в мессенджеры, где люди набирают текст на клавиатуре, лишая своего собеседника возможности увидеть свой почерк, уникальный для каждого, тем самым делая общение более быстрым и удобным, но в то же время менее душевным. Кроме того, многие люди из старшего поколения до сих пор отправляют друг другу открытки, так как считают такой способ общения более теплым. Тем не менее, с помощью современных технологий можно в какой-то степени перенести душевность бумажных писем, написанных от руки, на экраны смартфонов, которые есть почти у каждого. То есть хотелось бы иметь возможность сфотографировать рукописный текст, каким-то образом обработать его и отправить другому пользователю анимацию, показывающую написание этого текста. Тем самым создаётся впечатление "живого"письма. Однако, имеющиеся в открытом доступе решения либо слишком сложны для обычного пользователя, либо не обладают всем нужным функционалом. Именно поэтому мы решили создать приложение, которое должно быть простым и понятным для рядового пользователя, не требовать от него никаких лишний действий, при этом максимально точно передавать ментальный опыт, получаемый от бумажных писем и открыток.

1.1 Цель

Создание приложения, позволяющего по сделанной только что, либо уже существующей фотографии бумажного письма создать анимацию написания этого текста от руки. Кроме того, пользователи должны иметь возможность обмениваться этими анимациями и воспроизводить их в приложении.

1.2 Задачи

1) Создание Android приложения с базовым функционалом.

2) Разработка серверного программного обеспечения.

3) Доработка приложения, чтобы оно было способно воспроизводить анимационный файл.

4) Предобработка изображения, удаление теней и шумов, трансформация изображения, сфотографированного под углом в вид "сверху".

5) Разбиение текста на изображении на отдельные строки для повышения точности определения порядка анимации.

6) Разбиение строк на множество отдельных штрихов.

7) Сортировка штрихов в порядке их написания внутри одной строки и сортировка строк на итоговом изображении.

8) Отображение полученных пикселей в правильном порядке в файл, создание анимационного файла, передача файла обратно на основной код сервера.

2 Создание Android приложения.

Выполнили Демушкин Игорь и Чалкин Александр

2.1 Теория

Одной из самых сложных и объёмных частей нашего проекта было создание Android-приложения, способного делать снимки с камеры, а также выбирать фото из галереи, и затем отправлять его на сервер. Приложение очень сильно переделывалось по мере написания проекта. Первая версия, представленная на КТ1, была написана Демушкиным Игорем. Впоследствии большая часть приложения была переделана Александром Чалкиным для лучшей работы на современных Android-ах, также были переделаны многие визуальные моменты. Разберём последнюю версию приложения.

В проекте реализовано приложение на Android, которое позволяет пользователю сделать фотографию рукописного текста или выбрать эту фотографию из галереи, а затем отправить ее на сервер для последующей обработки. Затем на сервере проводится обработка фотографии - обнаружение текста и его выделение. После этого обработанное изображение отправляется пользователю и отображается в приложении.

Пользователь делает фото с помощью встроенного приложения камеры на устройстве. Были изучены различные методы по работе с камерой Android-устройства. Самым оптимальным для нашего проекта является внутренний вызов встроенного приложения камеры. Этот метод не требует работы с камерой на „низком уровне“, но в то же время не позволяет использовать какие-то специфичные настройки при запуске камерного блока на Android-устройстве. По той причине, что в нашем проекте на данный момент требуется делать обычную фотографию, было решено использовать именно этот метод вызова стандартного приложения камеры. В тоже время данная реализация предоставляет широкий спектр возможностей для самого пользователя. Ему становится доступны такие функции, как зум, вспышка и другие, а также разрабатываемое приложение становится совместимым для большего количества

Android-устройств, так как кастомная реализация камеры могла использовать тот функционал, который на отдельных девайсах недоступен. После того, как пользователь сделал фото, оно отправляется на локальный сервер посредством HTTP-соединения. Для этого используется стандартная Java- библиотека java.net.HttpUrlConnection. При отправке изображения создается POST request серверу, сервер его принимает и отправляет response. После получения POST запроса сервер начинает обработку информации.

2.2 Практика

Первой при включении приложения запускается функция onCreate, где сначала инициализируются наши переменные.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    mainView = findViewById(android.R.id.content);
    cameraButton = findViewById(R.id.cameraButton);
    galleryButton = findViewById(R.id.galleryButton);
    pictureReceived = findViewById(R.id.pictureReceived);
```

Теперь объясним, как работает функция, которая будет запускаться при нажатии на кнопку камеры ("снять фотографию").

Сначала мы проверяем, что в данный момент никакая картинка не отправляется. Это делается для того, чтобы не перегрузить наш сервер. Если какая-то фотография на данный момент отправляется, выводится соответствующее сообщение. Иначе мы создаём, грубо говоря, путь до временной фотографии photoUri, которая будет удалена после окончания работы нашего приложения. После этого уже запускаем встроенное приложение камеры, работа которого будет перехвачена с помощью cameraResultLauncher.

Теперь посмотрим, что будет происходить после обрабатывания приложения камеры. Если нам вернулся result = True, то мы создаём объект класса ImageProcess, и запускаем на нём метод processImage. Подробнее этот класс будет описан позже. Если нам вернулся result = False, то выводится соответствующее сообщение.

```java
cameraResultLauncher = registerForActivityResult(
        new ActivityResultContracts.TakePicture(),
        result -> {
            if (result) {
                ImageProcess imageProcess = new ImageProcess(
                        this,
                        mainView,
                        isImageSent,
                        photoUri
                );
                imageProcess.processImage();
                // TODO callback when the data from the server is retrieved
                // TODO also delete temp photo here
            } else {
                showSnackBar("No photo taken");
                isImageSent.set(true);
            }
        }
);
```

Посмотрим, как работает функция отвечающая за кнопку выбора фотографии из галереи.

В целом очень похоже на то, что было с кнопкой камеры. Сначала мы проверяем, что какая-то картинка отправляется, и если это не так, то выводится соответствующее сообщение. Иначе мы запускаем функцию, взятую из библиотеки для android, которая позволяет запустить приложение для выбора фотографии из галереи. Отметим также, что данный метод соответствует последним стандартам API, то есть версии Android, и даёт доступ к фотографиям из галереи для приложения без каких-либо дополнительных разрешений. Результат работы этой функции будет описан в следующем абзаце.

```java
galleryButton.setOnClickListener(view -> {
    if (!isImageSent.get()) {
        showSnackBar("зачилься другалёк, картинка отправляется");
        return;
    }
    isImageSent.set(false);
    photoPickerResultLauncher.launch(new PickVisualMediaRequest.Builder()
            .setMediaType(ActivityResultContracts.PickVisualMedia.ImageOnly.INSTANCE)
            .build()
    );
});
```

Теперь рассмотрим, что будет происходить после того, как пользователь выбрал фотографию из галереи. Если он выбрал валидную картинку, то есть uri != null, то создаётся, как и после снятия фотографии, объект класса ImageProcess, на котором будет запущен метод processImage. Если же пользователь выбрал невалидную фотографию, то будет выведено соответствующее сообщение.

```
photoPickerResultLauncher = registerForActivityResult(
        new ActivityResultContracts.PickVisualMedia(),
        uri -> {
            if (uri != null) {
                ImageProcess imageProcess = new ImageProcess(
                        this,
                        mainView,
                        isImageSent,
                        uri
                );
                imageProcess.processImage();

                // TODO callback when the data from the server is retrieved
            } else {
                Log.d("PhotoPicker", "No media selected");
                showSnackBar("No image selected");
                isImageSent.set(true);
            }
        });
```

Теперь посмотрим на функцию createPhotoUri, которая позволяет создать путь, по которому будет сохранена временная фотография.

```
protected Uri createPhotoUri() {
    File mediaStorageDir = getExternalFilesDir(Environment.DIRECTORY_PICTURES);
    File photoFile =
            new File(mediaStorageDir.getPath() + File.separator + "photo.jpg");
    return FileProvider.getUriForFile(
            this,
            BuildConfig.APPLICATION_ID + ".provider",
            photoFile
    );
}
```

Ещё покажем два маленькие функции onDestroy, которая вызывается при завершении нашей программы, и showSnackBar, которая выводит пользователю сообщения.

```
@Override
protected void onDestroy() {
    super.onDestroy();
    Log.e("onDestroy", "onDestroy is called");
    if (getContentResolver().delete(photoUri, null, null) == 0) {
        Log.e("FileDeletion", "Failed to delete photo");
    } else {
        Log.e("FileDeletion", "Photo deleted");
    }
}
```

```
protected void showSnackBar(String message) {
    FunctionClass.showSnackBar(mainView, message);
}
```

Теперь опишем те функции, которые написаны в файле ImageProcess.java.

Здесь написан класс ImageProcess, которым мы пользовались в MainActivity.java. Его главной функцией является processImage, где вызывается методы rotateBitmap, который переворачивает картинку, если это нужно, и sendImageToServer, который отсылает изображение на сервер. Обе этих функции определены в этом же классе ImageProcess.

```
public void processImage() {
    executor.execute(() -> {
        Bitmap rotatedBitmap = rotateBitmap(imageUri);
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
        rotatedBitmap.compress(Bitmap.CompressFormat.JPEG, 100, byteArrayOutputStream);
        byte[] image_bytes = byteArrayOutputStream.toByteArray();
        sendImageToServer(image_bytes);
    });
}
```

```
protected Bitmap rotateBitmap(Uri uri) {
    Bitmap bitmap = null;
    try {
        bitmap = BitmapFactory.decodeStream(context.getContentResolver().openInputStream(uri));
        ExifInterface exif = new ExifInterface(
                context.getContentResolver().openInputStream(uri));
        int orientation = exif.getAttributeInt(ExifInterface.TAG_ORIENTATION,
                ExifInterface.ORIENTATION_NORMAL);
        int rotationAngle = 0;
        switch (orientation) {
```

```
protected void sendImageToServer(final byte[] imageBytes) {
    new Thread(() -> {
        HttpURLConnection connection = null;
        try {
            URL url = new URL(SERVER_URL);
            connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("POST");
            connection.setDoOutput(true);
            connection.setRequestProperty("Content-Type", "image/jpeg");
            connection.setRequestProperty("Content-Length", String.valueOf(imageBytes.length));
            connection.setConnectTimeout(10_000);
            connection.setReadTimeout(30_000);
```

Также в этом классе определены функции getServerURL, которая возвращает URL сервера, указанного в файле config/config.yaml, и showSnackBar, которая выводит пользователю сообщение. Функция аналогична той, что была в MainActivity.java.

```
protected String getServerURL() {
    Properties properties = new Properties();
    try {
        assert context != null;
        InputStream input = context.getAssets().open("config/config.yaml");
        properties.load(new InputStreamReader(input));
    } catch (IOException e) {
        e.printStackTrace();
        showSnackBar("config error");
        throw new RuntimeException("Missing config");
    }
    return "http://"
            + properties.getProperty("ip_address")
            + ":"
            + properties.getProperty("port")
            + "/";
}

protected void showSnackBar(String message) {
    FunctionClass.showSnackBar(view, message);
}
```

3 Разработка серверного программного обеспечения.

Выполнил Кульпанович Данила.

3.1 Теория

Для начала следует понять, зачем нам нужен сервер. Во первых, основной моделью распространения приложений среди пользователей в данный момент являются приложения по типу Google Play (на платформе Android) или AppStore (на устройствах от компании Apple). В них действует модерация всех обновлений для приложений, которая иногда может идти от нескольких недель до месяца. Из-за этого, если мы внесем какие либо улучшения в алгоритмы обработки изображения или создания анимации, то пользователи не смогут сразу получить улучшенную версию, им придется ждать одобрения обновления модерацией интернет-магазина приложений. Во вторых, не у всех пользователей смартфоны достаточно производительны, чтобы быстро исполнять все компоненты программы, а также, на платформе android, есть множество смартфонов с различными моделями процессоров от различных производителей. Из-за этого повышается сложность исправления багов, ведь воспроизвести условия, при которых произошла ошибка, становится достаточно нетривиальной задачей. Так же в будущем не выйдет производить глубокие оптимизации кода, зависящие от какой либо части процессора (или иной вычислительной компоненты), ведь на некоторых смартфонах его может не быть. И в третьих, производительность серверного оборудования (даже если сервер развернут на обычном компьютере или ноутбуке) выше даже чем на самых новых и дорогих смартфонах, что позволяет значительно ускорить исполнение всех компонент программы. Потери времени, связанные с передачей файлов будут незначительны, если грамотно сократить объем передаваемых данных, о чем будет написано далее в работе.

Надо заметить, что работа любого приложения, использующего интернет соединение разделена на две компоненты - клиентскую и серверную. Клиент - это приложение на смартфоне пользователя, подробнее его работа описана в прошлой главе. Сервер - это то что описывается в этой главе.

Основа взаимодействия серверной и клиентской части - протокол передачи данный http, в работе будет использоваться один из основных его методов - POST запрос. Его суть состоит в том, что клиент отправляет на сервер некоторое число байт - в нашей случае это фотография, закодированная в определенном формате, а затем сервер отправляет в ответ на запрос также некоторую последовательность байт - в нашем случае файл с анимацией.

В итоге нам необходимо запустить сам сервер используя протокол http, создать обработчик POST запросов и уже в его теле запускать программы, отвечающие за обработку изображения и создание анимации.

3.2 Практика

Для написания серверной части воспользуемся языком Python и встроенной библиотекой http. Данный язык прост в использовании, и, что важно для нашего проекта, является основным для решения задач машинного и глубинного обучения благодаря широкому спектру готовых решений. Использование одинаковых языков для разных частей проекта позволит нам не создавать лишних переходов и каналов обмена данными, ускоряя работу программы и облегчая написание кода.

Ip адрес и порт, на котором мы открываем канал связи для сервера будет записан в yml конфигурационном файле, для упрощения тестирования работы сервера.

Итак, используя библиотеку http и её класс httpRequestHandler создадим сервер, способный обрабатывать запросы с клиентской части. Также заполним поле метода doPost, обрабатывающего входящие post запросы.

В любом отправленном наборе данных также включена заголовочная и замыкающая часть. В заголовке лежат данные, необходимые для верной трактовки полученных байт. В замыкающей части хранятся данные об авторе сообщения, что в нашем проекте не потребовалось, однако при добавление аналитического функционала (например, задачи определения текущей аудитории, число активных пользователей и тд.) эти данные пригодились бы. Для разбиения сообщения на заголовки, тело и замыкающую часть воспользуемся функционалом библиотеки requests.

После успешного получения байт из запроса определим тип файла, считав заголовок запроса. Затем считаем тело запроса, в котором и лежат данные полученного изображения. Из них через методы библиотеки OpenCV сформируем изображение обратно.

После этого начинается исполнение остальных компонент программы, результат работы которых - файл с анимацией. Заполним заголовок ответа, а также загрузим туда данные результирующего файла. После этого отправим клиенту код возврата 200 - результат успешного исполнения запроса, а также содержимое ответа.

4 Получение изображения выравненной бумаги из исходного изображения, сфотографированного под углом

Выполнил Матосян Александр

Перед тем как начать процесс детекции текста, надо выполнить некоторые действия с изображением, чтобы получать хорошие результаты при самом поиске рукописи. Первым

этапом этой предобработки является трансформация полученного изображения к виду "сверху", иными словами: мы хотим оставить на изображении лишь бумагу с рукописным текстом, как это делают сканнеры документов на телефонах.

4.1 Основные определения

Черно-белое изображение - это представление, в котором каждый пиксель изображения характеризуется своей яркостью - числом от 0 до 255. Чем больше значение, тем ярче пиксель, то есть 0 - черный, 255 - белый, а остальные значения - это оттенки серого.

Морфологические операции в обработке изображений представляют собой набор математических операций, применяемых к пикселям изображения с целью изменения их формы, размера. Обычно применяются к бинарным изображениям, где каждый пиксель может принимать одно из двух значений: черный или белый. Важные для нас морфологические операции :

1 Расширение (Dilation): Увеличивает объекты на изображении путем добавления пикселей к их границам. Это помогает заполнить пробелы и объединить близко расположенные объекты.

2 Эрозия (Erosion): Уменьшает объекты на изображении путем удаления пикселей из их границ. Она полезна для удаления шума, разделения сливающихся объектов и разрывов в объектах.

3 Открытие (Opening): Сочетание эрозии, за которой следует расширение. Используется для удаления шума и отделения объектов, которые слились.

4 Закрытие (Closing): Сочетание расширения, за которым следует эрозия. Используется для заполнения пробелов в объектах и слияния разрывов.

Размытие Гаусса - фильтр для изображения, сглаживающий (усредняющий) резкое изменения яркости соседних пикселей, позволяющий сделать изображение более равномерным по цвету. Вычисляется с использованием сверточной матрицы, применяемой к каждому пикселю изображения. Коефициенты матрицы вычисляются как значение из нормального распределения.

Алгоритм Кэнни обнаружения границ - это алгоритм обнаружения границ объектов на изображении. Основной идеей алгоритма является то, что пикселями границ объявляются те, в которых достигается локальный максимум градиента в направлении вектора градиента. Данный алгоритм реализован в библиотеке OpenCV в методе cv2.Canny().

4.2 Обнаружение листа бумаги

Первым этапом решения задачи, которой посвящена данная глава, является обнаружение листа бумаги. Как правило, письма пишут на бумаге прямоугольной формы, поэтому можно обобщить подзадачу: нужно обнаружить прямоугольник на изображении. Будем пытаться найти контур бумаги. Для начала немного подготовим изображение:

1 Переведем изображение в черно-белое

2 Применим размытие Гаусса, чтобы убрать мелкие шумы, которые в дальнейшем могут помешать обнаружению правильного контура

3 Применим морфологическую операцию закрытия, то есть последовательно применим сначала dilation, затем erosion. Таким образом размоются текст и другие объекты из фона

4 Применим алгоритм Кэнни для выделения граней объектов.

Теперь у нас есть бинарное изображение(пиксели белые/черные), в котором белым выделены грани объектов, и так как мы постарались избавиться от шумов и других мешающих факторов, то на этом изображении в большинстве случаев есть грани бумаги, а также мало граней других объектов. Вернемся к форме бумаги: грани бумаги - это 4 прямые, поэтому уже на бинарном изображении будем искать линии. Поиск линий будем производить с помощью алгоритма LSD[Статья с описанием алгоритма поиска прямых]. Полученные линии разделим на горизонтальные и вертикальные, по следующему критерию:

$$| x_2 - x_1 | > | y_2 - y_1 | \tag{1}$$

Где x_1 y_1 и x_2 y_2 - координаты концов прямой.

Также, для каждой группы создадим пустой холст, на котором отобразим линии из соответствующей группы, но чуть длиннее и толще. Зачем так делать? Дело в том, что алгоритм поиска линий может найти не всю линию, либо найти несколько её отрезков, поэтому мы и рисуем линии на отдельных холстах утолщая и удлиняя их, чтобы получить связанную компоненту и выделить из неё всю прямую. Теперь, на каждом из холстов выделим контуры получившихся компонент. В каждом контуре найдем минимальную и максимальную координату x: X_min и X_max. Затем найдем все точки из контура вида (X_min, y) и пусть Y_avgmin - среднеарифметическое координат y этих точек, тогда точка (X_min,

Y_avgmin) будет одним из концов аппроксимируемой прямой. Сделаем аналогичные действия для X_max и получим другой конец - (X_max, Y_avgmax). Зная прямые, можем найти потенциальные края бумаги. Потенциальными краями бумаги будем считать два типа точек:

1 Точки соответствующие концам найденных прямых

2 Точки пересечения горизонтальных и вертикальных прямых

Концы найденных прямых сразу добавим в множество потенциальных краев, а чтобы найти пересечения горизонтальных и вертикальных прямых сделаем следующее: на двух пустых холстах(со значениями пикселей равными нулю) нарисуем отдельно горизонтальные прямые и вертикальные прямые, причем толщину прямых сделаем в 1 пиксель и яркость пикселей тоже 1. Сложив оба холста попиксельно, получим что точки пересечения - это пиксели с яркостью равной 2. Добавим эти точки в множество потенциальных краев бумаги. Чтобы сузить область поиска, отфильтруем множество: уберем точки, которые находятся очень близко к другим.

На данный момент имеем множество потенциальных вершин углов бумаги. Из него надо найти четыре точки, которые соответствуют вершинам бумаги. Рассмотрим все четверки из множества потенциальных вершин. Каждую четверку точек отсортируем по часовой стрелке и проверим на соответствие следующим свойствам:

- Площадь прямоугольника, построенного на этих точках, больше или равна 25% площади изображения. Это проверка важна, так как бумага скорее всего занимает много места на изображении.

- Каждый угол прямоугольника примерно равен 90 градусам. Сами углы ищем с помощью формул из аналитической геометрии

Если четверка не удовлетворяет этим условиям, то не рассматриваем ее дальше. Отсортируем оставшиеся четверки по убыванию площади прямоугольника, построенного на них, и оставим лишь несколько первых четверок. Теперь у нас есть несколько наборов точек, которые соответствуют самым большим четырехугольникам на изображении. Из этих четырехугольников надо выбрать тот, который наиболее близок к прямоугольнику. Как это сделать? Воспользуемся свойством из школьной геометрии: диагонали параллелограмма равны. То есть, самым подходящим для нас четырехугольником является тот, у которого модуль разности диагоналей наименьший. Поэтому искомым набором вершин углов бумаги будет четверка точек,

четырехугольник которых имеет наименьший модуль разности диагоналей. Таким образом, вышеописанный алгоритм находит четыре точки, соответствующие вершинам углов бумаги.

4.3 Преобразование фотографии, снятой под разными углами, для получения выровненного изображения бумаги

Теперь мы знаем расположение бумаги на изображении - нам известны четыре вершины ее углов. Имея эту информацию можем сделать преобразование фотографии, в результате которого получим выровненное изображение бумаги. Это можно сделать с помощью преобразования плоскости, называемого гомографией, которое выражается равенством:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = H \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{2}$$

где x, y – исходные координаты точки, x, y – координаты после преобразования, H – матрица преобразования размера 3×3. Для нахождения этой матрицы достаточно знать 4 точки из исходной плоскости и 4 соответствующие конечные точки. Сама матрица H ищется с помощью решения системы линейных уравнений. В нашем случае исходными точками являются найденные вершины углов бумаги, а конечными точками являются точки углов прямоугольного изображения. Для программной реализации нахождения этой матрицы можно использовать метод getPerspective из библиотеки OpenCV. А чтобы применить найденную матрицу к исходному изображению используем метод warpPerspective из той же библиотеки. Таким образом получили желаемый результат - изображение выровненной бумаги.

4.4 Реализация

Реализация всех этих функций представлена в классе Scanner. Главной функцией, где происходит обнаружение листа бумаги и его трансформация, является метод scan. Внутри метода scan вызывается функция get_contour. В ней ищутся контуры бумаги на изображении. Большинство функций, определённых в классе Scanner, как раз относятся к методу get_contour, вызываемых по мере работе последней. Поиск потенциальных вершин углов бумаги происходит в методе get_corners, она вызывается из метода get_contour. После того как метод get_contour возвращает вершины углов бумаги, вызывается метод four_point_transform. В нем происходит вычисление матрицы гомографии, которая применяется к исходному изображению, снятого под углом, и преобразовывает его в изображение выровненной бумаги. Уже

после окончания работы функции four_point_transform под конец происходят незначительные преобразования сконструированной фотографии.

5 Предобработка изображения, удаление теней и шумов. Подготовка изображения к детекции участков с текстом

Выполнил Матосян Александр

Для получения хороших результатов при детекции текста нужно провести некоторую предобработку полученного изображения. И одним из важнейших этапов этой предобработки является удаление теней, шумов и других артефактов изображения. Если до этого этапа мы сумели получить изображение выровненной бумаги, то мы уже избавились от шумов на фоне исходного изображения, и остается избавиться от артефактов на бумаге. Если же найти бумагу не удалось и мы работаем с исходным изображением, то надо постараться минимизировать количество теней и других шумов на всём изображении. Зачем вообще нужно чистить изображение от шумов? Такая обработка нужна, чтобы на бинаризированном изображении остался только текст и на этапе детекции текста не распозналась какая-то тень за часть текста.

5.1 Алгоритм обработки

Для начала переведем изображение в черно-белое. На таком изображении пиксели в участках с тенями будут иметь значения близкие к середине диапазона яркости, а нужный нам текст ближе темный, поэтому пиксели букв будут иметь значения ближе к началу диапазона. Затем применим размытие Гаусса с большим ядром Гаусса, чтобы получить очень размытую картинку. Поэлементно поделив черно-белое изображение на размытое с коэффициентом 255, мы поднимем яркость теневых участков при этом текст останется темным. Далее бинаризуем полученное изображение, то есть яркие пиксели сделаем белыми, а темные черными. Для вычисления порога бинаризации используем метод Оцу. Так как мы постарались повысить яркость теневых участков, то на бинаризированной картинке тени уйдут в белый цвет, а текст в черный. В конце применим морфологическую операцию dilation, чтобы восстановить потерянные элементы некоторых букв. Итого, получили бинарное изображение без теней и минимальным шумом.

5.2 Реализация

Предобработка изображения реализована на языке Python3 с использованием библиотеки OpenCV. Код вынесен в отдельную функцию. Из библиотеки OpenCV использованы методы:

1. cv2.GaussianBlur() - для Гауссовского размытия
2. cv2.divide() - для поэлементного деления. Принимает две массива и численный параметр scale, для домножения элементов на этот параметр
3. cv2.threshold() - для бинаризации. Данный метод вызывается с параметром cv2.THRESH_OT чтобы порог бинаризации вычислялся по алгоритму Оцу

6 Разбиение текста на изображении на отдельные строки для повышения точности определения порядка анимации.

Выполнил Демушкин Игорь

6.1 Теория

Основная проблема сортировки штрихов заключается в том, что сортировать вместе штрихи со всего изображения так, чтобы написание текста выглядело правдоподобным, невозможно. Сортировка штрихов по x-координате, y-координате или по какой-то линейной функции от этих двух координат не даст хорошего результата. Это происходит из-за того, что человек пишет текст "по-особенному сначала внутри строки слева направо, а потом следующую строчку снизу от предыдущей. Отдельно стоит отметить, что в разных культурах и разных языках способ письма различается. Именно поэтому для других языковых групп, где способ письма отличается от латинских стандартов (например, арабский язык с его характерной арабской вязью, когда люди пишут буквы справа налево), надо использовать особую сортировку штрихов и строк. В нашем же случае сортировка строк происходит по y-координате и вторым приоритетом по x-координате, то есть первой будет считаться строка, распложенная выше всех, а если таких строк несколько, то будет взята самая левая строка. Стоит отметить, как вообще происходит разбиение текста на различные строки. Для этой цели было решено использовать библиотеку OpenCV. С помощью различных её методов, главным образом erode и dilate, создаётся отдельное чёрно-белое изображение с белыми пятнами, которые показывают места, где написаны отдельные строчки, и пространство вокруг них. Далее по этим белым пятнам создаются маленькие изображения, отвечающие каждое за одну строку. Также запоминается расположение этих фотографий относительно исходной, чтобы в конце обработки мы смогли построить правильную анимацию. Дальнейшая работа (разбиение на штрихи и т.д.) будет происходить как раз в каждом изображении по отдельности.

6.2 Практика

Теперь поподробнее напишем, как работает разбиение текста на изображении на отдельные строки.

Первым делом считываем изображение с помощью OpenCV, далее преобразуем его в чёрно-белый вариант для лучшей обработки текста. После переведём фотографию в бинарный вид.

```
im = cv.imread('img.jpg')
imgray = cv.cvtColor(im, cv.COLOR_BGR2GRAY)
ret, thresh = cv.threshold(imgray, 0, 255, cv.THRESH_OTSU |
cv.THRESH_BINARY_INV)
```

Далее применяем два метода: erode, который используется для удаления небольшого шума в виде отдельных небольших точек, и dilation, который в свою очередь размывает изображение. Метод dilate мы определили так, что он размывает очень сильно по горизонтали и слабо по вертикали. Таким образом одна строка, где символы расположены достаточно близко к друг другу, превращается в одно единое белое пятно.

```
rect_kernel_for_erosion = cv.getStructuringElement(cv.MORPH_RECT, (1, 1))
rect_kernel_for_dilation = cv.getStructuringElement(cv.MORPH_RECT, (18, 1))
erosion = cv.erode(thresh, rect_kernel_for_erosion, iterations = 3)
dilation = cv.dilate(thresh, rect_kernel_for_dilation, iterations = 3)
```

Далее напишем BFS, который разобьёт наши пиксели на компоненты связности, тем самым мы поймём, какой пиксель к какому белому пятну относится.

Первым делом инициализируем двумерный массив meeted.

```
meeted = []
for i in range(len(dilation)):
    meeted.append([])
    for j in range(len(dilation[0])):
        meeted[i].append(0)
```

Далее уже собственно пишем BFS, не буду пояснять работу этого алгоритма, только отмечу, что два пикселя будут считаться соседними, то есть связанными друг с другом, если они имеют общую сторону. То есть обычно пиксель будет связан с четырьмя другими.

```
count_for_bfs = 100
count_for_img = 1
strings = []
for i in range(len(dilation)):
    for j in range(len(dilation[0])):
        if (int(dilation[i][j]) == 255 and meeted[i][j] == 0):
            first_coord = [i, j]
            second_coord = [i, j]
            count_for_bfs = max(1, (count_for_bfs + 100) % 254)
            q = queue.Queue()
            q.put((i, j))
            meeted[i][j] = count_for_img;
            strs[i][j] = count_for_bfs # для дебага
            while not q.empty():
                cur_i, cur_j = q.get()
                for shift in [[0, -1],[1, 0], [-1, 0], [0, 1]]:
                    new_i = max(0, min(len(dilation) - 1, shift[0] + cur_i))
                    new_j = max(0, min(len(dilation[0]) - 1, shift[1] + cur_j))
                    if (int(dilation[new_i][new_j]) == 255 and meeted[new_i][new_j] == 0):
                        if (first_coord[0] > new_i):
                            first_coord[0] = new_i
                        if (first_coord[1] > new_j):
                            first_coord[1] = new_j
                        if (second_coord[0] < new_i):
                            second_coord[0] = new_i
                        if (second_coord[1] < new_j):
                            second_coord[1] = new_j
                        q.put((new_i, new_j))
                        meeted[new_i][new_j] = count_for_img
                        strs[new_i][new_j] = count_for_bfs

            cropped = im[first_coord[0]:second_coord[0] + 1, first_coord[1]:second_coord[1] + 1]
            for ii in range(len(cropped)):
                for jj in range(len(cropped[0])):
                    if (meeted[ii + first_coord[0]][jj + first_coord[1]] != count_for_img):
                        cropped[ii][jj] = 255;
            strings.append(cropped)
            cv.imwrite('img' + str(count_for_img) + '.jpg', cropped)
            count_for_img += 1
```

Во время обработки одной компоненты связности мы ищем левую верхнюю координату и правую нижнюю таким образом, чтобы вся компонента связности лежала внутри прямоугольника, образованного этими двумя координатами. Далее мы создаём новое изображение, являющееся вырезанным прямоугольником из исходной фотографии. После этого зануляем те пиксели (точнее делаем равными 255, так как белый цвет соответствует этому числу) на вырезанном изображении, которые не принадлежат нашей компоненте связности, которую в данный момент обрабатываем. Полученное изображение добавляем в массив strings.

На выходе мы получаем массив strings, где лежат отдельные картинки, где каждая фотография содержит одну строку.

7 Разбиение строк на множество отдельных штрихов.

Выполнил Чалкин Александр

7.1 Теория

Штрихами будем называть отдельные непрерывные линии, которые пользователь рисовал, не отрывая ручки от бумаги. Задача выделения штрихов на изображении обычно изучается в контексте распознавания текста. Выделяется 2 вида распознавания - онлайн и оффлайн. Отличаются они тем, что в онлайн методе доступна "временная"информация о том, каким образом написан текст - например, порядок и скорость написания. Стоит отметить, что онлайн системы распознавания обычно добиваются большей точности определения. Один из методов решения задачи оффлайн распознавания можно выделить трансформирование задачи из оффлайна в онлайн. Для этого необходимо получить некоторую временную информацию - то есть то, как был написан текст на картинке. Можно, к примеру, разбить текст на отдельные штрихи, затем отсортировать их в некотором порядке, в некотором смысле точно приближающем реальное написание текста. Тогда можно использовать эту информацию для превращения задачи оффлайн распознавания текста в онлайн. Мы же будем использовать ее для симуляции написания текста.

В настоящее время можно выделить 2 главных подхода к решению задачи выделения штрихов на изображении: эвристический и метод машинного обучения. В первом случае предлагается локальный анализ текста, основанный на некоторых эвристиках - общее предпочтение письма слева направо и сверху вниз, а также упор на минимальном угле отклонения (в точках соприкосновения нескольких штрихов) ручки во время письма, а также минимальном количестве отрывов ручки от листа, так как это занимает лишнее время во время письма. Второй метод, ставший популярным относительно недавно, основан на алгоритмах глубинного обучения, а именно на использовании сверточных нейронных сетей. Модель обучается на базах картинок с некоторым рукописным текстом и на выходе дает упорядоченный массив из штрихов. В данном проекте используется эвристический метод из-за его легкости в сравнении с подходом, основанном на машинном обучении.

Также стоит отметить, что вышеупомянутая задача выделения штрихов также изучается в контексте верификации подписей и распознавания восточноазиатских иероглифов. В первом как подпись рассматривается небольшое изображение, а обработка картинок с большим разрешением делает алгоритмы невалидными, так как они работают слишком долго. Иероглифы сильно отличаются от других видов письма, так как они имеют свои определенные своды правил написания, не всегда верные для западной рукописи.

7.2 Практика

На вход подается бинарное изображение, содержащее строку рукописного текста. Сначала проводится "скелетизация"изображения - то есть его утончение до изображения шириной в 1 пиксель. Затем скелет некоторым образом разбивается на штрихи, после чего определяется направление написание каждого штриха.

Для скелетизации изображения используется разновидность алгоритма Чжана и Суэна, а именно алгоритм Вона и Чжана, так как он лучше сохраняет форму диагональных штрихов.

Затем строится графовое представление скелета изображения. Скелет разбивается на "соединения"и "отрезки". Черный пиксель, имеющий в своей окрестности (ближайшие 8 соседей) ровно 2 других черных пикселя (причем соседи не должны быть соседями друг для друга) будет считаться частью пикселя-отрезка. Все другие черные пиксели назовем пикселями-соединениями. Ими, в основном, будут являться пиксели, имеющие 3 или более соседей, что, скорее всего, означает, что по этому месту на листе бумаги ручка проходила более одного раза. Но также пикселями-соединениями являются отдельные пиксели и пиксели, имеющий только одного соседа. Связный набор пикселей-отрезков назовем "отрезком а пикселей-соединений - соединением. Тогда каждый набор пикселей в отрезке можно упорядочить естественным образом - каждый следующий пиксель будет соседом предыдущего. Причем если первый пиксель является соседом последнего, то отрезок является замкнутой петлей. Такой отрезок будем считать не отрезком, а соединением. Таким образом, каждый отрезок ограничен соединениями - то есть имеет начало и конец. А значит такой набор из соединений и отрезков представим в виде графа - соединение является вершиной, а отрезок - ребром. Тогда путь в таком графе является одним из способов написать предложенный для анализа текст на листке бумаги.

При таком подходе разбиения пикселей и из-за использованного алгоритма скелетизации появляется проблема - многие соединения появляются из-за неточности алгоритмов, а это может ухудшить качество анимации. Для решения этой задачи предлагается эвристическое правило - убрать из графа каждый отрезок, имеющий длину, меньшую чем $C \cdot w_{avg}$, и соединить его концы. w_{avg} - средняя ширина штрихов всего текста, C - некоторая константа. Такой подход сработает, так как средняя ширина штрихов рукописного текста обычно является постоянной. w_{avg} вычисляется как средняя по всем отрезкам минимальная ширина по четырем направлениям (горизонтальном, вертикальном, диагональных) из каждого пикселя.

Для получения порядка написания штриха используются следующие эвристики:

- количество штрихов предполагается минимальным
- Разность в направлении написания внутри штриха предполагается минимальной

При написании некоторых букв (например, р, ш, ж) по одному и тому же отрезку ручка проходит несколько раз. На таких примерах наш алгоритм будет создавать слишком много штрихов. Для решения этой проблемы найдем общие отрезки и соединим их с разделенными штрихами. Общими отрезками будут считаться:

- отрезки, имеющие две различные конечные точки, которые являются вершинами в графе с нечетной степенью
- отрезки, обе конечных точки которых также являются конечными вершинами пути в графе, и угол между ними не близок к $\frac{\pi}{2}$.

8 Сортировка штрихов в порядке их написания внутри одной строки и сортировка строк на итоговом изображении.

Выполнил Демушкин Игорь

8.1 Теория

Это довольно несложная задача, но она имеет большое значение для правильного порядка анимации штрихов. Так как мы заранее разбили текст на изображении на отдельные строки, то надо сначала понять, в каком порядке анимировать штрихи внутри одной строки. Это будем делать максимально примитивным образом, сортируем штрихи по x-координате, причём в качестве координаты штриха будем рассматривать его начальную точку, то есть точку, откуда он будет анимироваться. Также с небольшим весом при сортировке будем учитывать y-координату. Это делается для того, что если штрихи находятся далеко друг от друга по высоте, скорее всего они были написаны в разное время. Также возможен случай, когда две строки на входном файле были написаны близко к друг другу таким образом, что некоторые буквы пересекаются с другими, которые относятся к иной строке. Тогда наше разбиение на строки "склеит"эти две строки, и итоговая анимация будет не совсем корректной.

Мы поняли, как сортируются штрихи внутри одной строки. Объясним сейчас, как сортируются сами строки. Строки тоже будем сортировать самым простым, но хорошо работающим, способом. Сначала сортируем по y-координате и потом по x-координате, то есть первой будет считаться строка, распложенная выше всех, а если таких строк несколько, то будет взята самая левая строка.

9 Отображение полученных пикселей в правильном порядке в файл, создание анимационного файла, передача файла обратно на основной код сервера.

Выполнил Кульпанович Данила

9.1 Терминология

Объем или вес файла - число байт, занимаемых файлом в постоянной памяти устройства, в контексте работы зачастую употребляется не абсолютной, а в оценочной характеристике (большой, малый и тд.).

Кодировка - создание целого файла по каким либо данным.

RGB(A) - вид хранения пикселя, при котором хранится 3 канала (зачастую целые значения от 0 до 255), соответствующие красному, зеленому и синему цвету, при смешение которых получается результирующий цвет. Также дополнительно можно хранит альфа значение, определяющее прозрачность (точная формула наложения есть далее в работе).

Координаты пикселя - размер отступа от левого верхнего угла изображения.

Файловый дескриптор - номер канала передачи данных, по которому можно передавать байты. К контексте работы - конкретно в контексте потока записи данных в постоянную память в конкретный файл, заранее созданный, но не заполненный данными.

BEBR - случайно выбранная комбинация букв, которая означает расширения файла, в котором будет хранится анимационный файл.

Расширение файла - то же, что и тип файла.

9.2 Теория

Для начала введем метрики, по которым сможем определить качество файла для решения нашей задачи. Во первых, файл должен иметь минимальный объем, но при этом

вообще не иметь, или иметь неразличимые потери в данных для пользователя. Это важно как для экономии места у конечного пользователя, так и для повышения скорости передачи данных в условиях, когда интернет соединение медленное или нестабильное. Во вторых, кодировка не должна отнимать много времени, ведь помимо данной части в проекте есть иные операции, и в совокупности может получится так, что пользователь не захочет пользоваться слишком медленным приложением.

Рассмотрим каждую задачу по отдельности. Для минимизации веса файла нам необходимо подробнее рассмотреть, какие данные мы получаем на вход, какие данные ожидаются как результат работы, а также, в каком виде пользователи будут смотреть анимацию. Результатом работы предыдущих частей программы является данные о ширине и высоте изображения, а также упорядоченный массив координат точек. Единственное, что мы можем сделать на основании этой информации - выбрать тип хранения и воспроизведения итогового файла. Существует 2 вида графических файлов - растровый и векторный. Растровый формат подразумевает, что изображение (в том числе и анимация) хранится как массив данных о каждом пикселе, которые затем отображаются в изображение по строкам (1 пиксель массива - 1 пиксель первой строки изображения). Заметим, что подобный формат близок к формату входных данных. Второй формат - растровый, изображение хранится в виде векторов (то есть неких кривых линий). Этот формат хорош тем, что четкость картинки не изменится при приближении, в растровом формате станут видны отдельные пиксели (т.к пиксель имеет фиксированные размеры, часто говорят о плотности пикселей на дюйм). Однако, с учетом входных данных нам будет необходимо произвести перекодировку массива пикселей в вектора, причем не обязательно простых форм (ведь подчерк у всех разный), а это может отнять много времени и занять много памяти. Кроме того, пользователи будут смотреть анимацию на смартфоне с небольшим экраном, а также же вряд ли будут приближать изображение, ведь текстовая анимация ценна только при взгляде на весь текст, отдельные фрагменты не интересны. И из этого следует, что нам подходит растровый тип файла, на который будут выводится по порядку пиксели из исходных данных, ведь у векторного есть недостатки, его преимущества в нашей задаче не важны, а недостатки растрового наоборот не повлияют на восприятие анимации.

Обратим внимание и на общий вид результирующей анимации. Она представляет из себя черный текст на белом фоне. Сразу заметим, что форматы для хранения различных видео нам не подойдут в силу избыточности веса - хранится в том числе и звук, кроме того в них них избыточные цветовые палитры - а нам нужны всего 2 цвета. Остается единственный вариант - файлы по типу gif, иногда именуемые как раз анимациями. По факту они из

себя представляют набор изображений, последовательно выводимых на экран, при этом достаточно быстро, чтобы пользователю казалось, что изображение плавно движется. Однако и тут можно произвести оптимизацию веса файла. Обратим внимание на то, что зачастую (кроме переноса строки) новые пиксели на изображении появляются достаточно сгруппировано, то есть рядом и почти в одном месте, что следует из самого процесса написания текста, рука почти не отрывается от бумаги, движется в определенной последовательности. Значит, для экономии места мы можем хранить не целиком каждое изображение, а только область с новыми пикселями. Для этого введем новые обозначение, а также некоторые формулы.

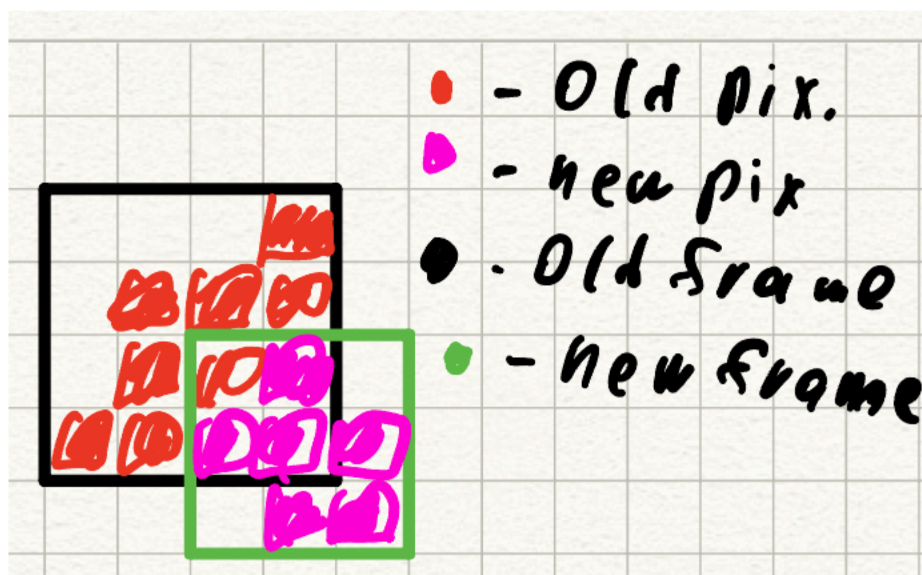
$$Xoffset = \min x_i, Yoffset = \min y_i \tag{3}$$

Где x_i, y_i - все координаты пикселей, выводимых в данном кадре, $Xoffset$ и $Yoffset$ - отступы (в пикселях) от левого и верхнего края изображения соответственно.

$$newheight = \max x_i - \min x_i, newwidth = \max x_i - \min y_i \tag{4}$$

Где $newheight$ и $newwidth$ - высота и ширина нового изображения.

Теперь мы сможем хранить не изображение в полном разрешении, а только ту его часть, которая изменилась. Однако, теперь возникла проблема наложения новый пикселей на старые (пример на изображении):



Решить её можно добавлением в палитру изображения нового цвета - полностью прозрачного (значение альфа канала равно нулю), а для того, чтобы фон остался белым самый первый кадр будет полного разрешения, при этом все не черные пиксели (фон) - белые.

Чтобы первый кадр действительно обновлял весь фон на белый, а последующие не накладывались друг на друга введем также 2 режима наложения - замена всех пикселей на новые, и смешение пикселей с учетом прозрачности. Оно будет выполнено по формуле:

$$R = B \cdot (1 - A) + F \cdot A \tag{5}$$

Где R - результат, A - значение альфа канала нового пикселя, B и F - значения старого и нового пикселя соответственно.

Дальнейшую оптимизацию веса можно было бы проводить так: хранить не изображения, пусть и малого разрешения, а только координаты пикселей. Однако в таком случае возрастет время на раскодировку изображения, а оно проводится на смартфоне пользователя, менее мощном, нежели сервер.

Теперь выберем метод хранения пикселей. Очевидно, что полный RGBA будет избыточен. В связи с тем, что у нас на каждом кадре финальной анимации всего 3 цвета - выберем палитровую (то есть каждому цвету заранее выдается свой номер), причем 2 битовую (1-битовая будет недостаточна) - то есть всего 4 варианта цветов (0 1 2 3 в двоичной системе счисления). Данная оптимизации позволит дополнительно сократить вес каждого кадра примерно в 16 раз (RGBA - 8 бит на каждый из 4 каналов).

Кроме того, для оптимизации объема файла можно применить сжатие изображения (зачастую выполняется построчно). Однако в документации по ссылке было доказано, что применение сжатия к изображениям с палитровой системой хранения цвета зачастую приводит лишь к увеличению объема файла.

Кроме того, мы хотим обозначать, какое время пройдет между кадрами (зачастую используется метрика число кадров в секунду), очевидно что это будут доли секунды, иначе анимация не будет плавной, пропадет иллюзия рукописного написания. Для этого будем хранить числитель и знаменатель дроби, означающей время в секундах отведенное на кадр (не в виде числа с плавающей точкой для избежания потери точности).

Теперь заметим, что данный формат достаточно близок к уже существующему, пусть и официально не принятому комиссией по стандартизации png формату хранения - к apng. Однако в apng есть некоторые дополнительные поля, а именно возможность указать стандартный, то есть базовый кадр, некоторые особые режимы наложения, возможность кодировать кадры разными палитрами, а также обязательное применения сжатия (чего мы, как пояснялось выше, не хотим делать). Заметим, что эти доработки к файлу не займут много дополнительного места (всего порядка нескольких байт) в итоговом файле, при этом избы-

точные цветовые палитры и сжатие можно обойти, написав собственный кодировщик для анимационного файла.

Итого, в каждом кадре будем хранить: байтовые данные самого изображения, его разрешение, отступы от краев, числитель и знаменатель времени кадра, режим наложения, а также же номер кадра, который нужен для возможности восстановить анимацию обратно по набору изображений. Дополнительно добавим поля, существующие в apng формате, но несущественные для нас.

Кроме того, сохраним финальное разрешение изображения и его цветовую палитру, но хранить их будем для всей анимации, а не для каждого кадра, благодаря чему сэкономим немного объем файла.

9.3 Практика

Для написания кода данной части проекта используется язык GO, в связи с тем что он достаточно быстр, в нем нету проблем с утечками памяти из-за неправильной работы с памятью, а также же потому что в нем есть крайне гибкая и удобная в использовании библиотека для работы с изображениями.

Код программы разбит на 6 файлов, вместе образующих единый комплекс, пакет в GO терминологии. Рассмотрим каждый из них поподробнее:

В файле constants.go лежат числовые значения необходимые как для упрощения написания кода (замена чисел на понятные обозначения), так и для тестирования работы отдельных частей программы. Введены обозначения для различных цветовых палитр а также алгоритмов сжатия, константы зачастую использовались для проведения тестов, в финальном коде используется лишь часть из них. Также сохранено строковое значение заголовка файла изображения типа PNG, опять же, для упрощения написания кода.

В файле frame.go объявлены две структуры, описывающие сам анимационный файл как набор кадров, а также каждый кадр в отдельности. Все поля были введены в теоретической части, только в качестве байтов изображения используется объект типа Image из стандартной GO библиотеки для работы с изображениями.

Теперь рассмотрим остальные файлы, в них уже находятся функциональные части, осуществляющие то или иное действие.

Начнем с основного файла, с которого начинается исполнения - main.go. Программа считывает из аргументов запуска имя файла, в котором хранится входная информация. Далее программа считывает все числовые значения из файла, причем первые 2 - ширина и

высота изображения, сохраняются отдельно, а остальные числа - координаты, переносятся в массив объектов типа точка - то есть структур с двумя числовыми значениями, координатами по x и по y. Вычисляем число пикселей на кадр, дабы анимация была реалистичной, с оптимальной скоростью написания текста. После этого создается объект типа BEBR, в который мы и будем записывать нужную информацию. Дальнейший код будет выполнен для каждого кадра. Число кадров вычисляется просто: разделим число пикселей на число пикселей на кадр, с округлением вверх.

С помощью библиотеки image создаем новое изображение с заданной цветовой палитрой. Ширина и высота первого кадра получены из файла (это ширина и высота результирующей анимации), для остальных кадров они вычисляются по формулам для newheight и newwidth из блока с теорией. Кроме того, первый кадр заполним белыми пикселями, в остальных фон по умолчанию прозрачный - 1 цвет в нашей палитре. Затем отрисуем новые пиксели на изображении. Заметим, что новые координаты будут вычисляться по формулам:

$$newX = x - Xoffset, newY = y - Yoffset = \min y_i \tag{6}$$

ведь изображения изначально смешенные. Дополнительно заполним оставшиеся поля кадра, все значения и формулы были описаны в теории. После заполнения структуры BEBR всеми кадрами создадим по средствам библиотеки os файл с уникальным названием, так как одновременно использовать приложение может несколько пользователей. Затем закодируем данные структуры BEBR в байтовую последовательность, функцией ENCODE уже из другого файла.

Оставшиеся 3 файла плотно взаимосвязаны, разделение либо функционально-смысловое, либо типовое (по типам данных), рассмотрим их все вместе.

Функция ENCODE по файловому дескриптору (открытому в main.go) и структуре BEBR кодирует в структуру типа encoder, из которой уже идет сохранение байт в файловый дескриптор. У самого encoder'а есть как поля, в которых хранятся общие данные анимации, так и временный буфер (созданный для сокращения обращений к постоянной памяти, более медленной, чем оперативная), а также отдельные поля для построчного сжатия изображений, нужные на этапе тестирования программы, но по факту не используемые в развернутом на сервере варианте. После заполнения полей структуры кодировщика функция Encode вызывает отдельные функции для записи каждого блока в память (как общих, так и индивидуальных блоков для каждого изображения). Все эти функции, начиная с write, схожи, ведь по факту просто переносят поля для каждого блока файла во временные буфера

и вызывают иные функции.

Однако становится ясно, что вечно записывать байты в буфер нельзя, он переполнится. Именно поэтому создана функция записи на файловый дескриптор чанка - то есть некоего блока данных, названная writeChunk. Она создает заголовок и окончание блока, по которому декодеровщик на стороне клиента поймет, к какой части файла принадлежит данный блок а также где он оканчивается.

Теперь обратим внимание на то, что в каждом кадре (структура Frame) значительно наибольший вес (то есть вес превосходит сумму всех остальных весов в несколько раз) имеет структура Image, хранящая непосредственно данные об изображении. Также ясно, что из-за веса все данные не влезут во временный буфер, хотя для остальных блоков его хватало. Именно поэтому для записи данных об image существует отдельная функция writeImage. Кроме непосредственно записи байтов картинки (построчно) сразу в файловый дескриптор эта функция способна применять алгоритмы сжатия для проведения тестирования эффективности сжатия к каждой строке, а также осуществлять построчную запись во временный буфер, отдельно созданный для сжимаемых строк, откуда затем идет запись в файловый дескриптор, что пригодится для случаев, когда для некоторых строк эффективны разные алгоритмы сжатия.

После записи всех данных, необходимых для анимации, на файловый дескриптор запишем также нуль-терминирующий байт, чтобы декодеровщик понял, где конец файла.

Следует также указать, что функция кодирования способна получать ошибку на любом этапе записи, и возвращать её как результат работы, это помогает избежать отправки битых файлов обратно пользователю.

После записи всех данных в файл его файловый дескриптор закрывается, а сам файл считывается в основном коде сервера, и его байта передаются как тело ответа на POST-запрос обратно пользователю, после чего декодеровщик в клиентском приложении воспроизводит анимацию на экране.

10 Заключение

В процессе разработки приложения "Warm Letters"были опробованы различные методы выделения составных элементов букв, предобработки изображения для упрощения этого процесса, а также определения порядка написания (вывода на экран), совпадающего или максимально точно приближенного к человеческому, выбраны наиболее оптимальные методы решения каждой задачи. Кроме того, составлены критерии, по которым можно миними-

зировать объем анимационного файла без потери его качества относительно видео высокого разрешения.

В результате было создано приложение, достаточно простое для использования любым пользователем, с лаконичным, но в то же время достаточным функционалом. Для работы программе не требуется производительный смартфон. В то же время, создание анимации работает достаточно быстро, не создавая пользователю неудобств в виде долгого ожидания результата.

Список литературы

[1] PNG Documentation

[2] HTTP Python module

[3] Requests

[4] Hypertext Transfer Protocol

[5] Go image package

[6] bufio package

[7] Activity Result API

[8] T. Y. Zhang and C. Y. Suen, “A fast parallel algorithm for thinning digital patterns,” Commun. ACM, vol. 27, no. 3, pp. 236–239, Mar. 1984.

[9] P. S. P. Wang and Y. Y. Zhang, “A fast and flexible thinning algorithm,” IEEE Trans. Comput., vol. 38, no. 5, pp. 741–745, May 1989.

[10] Text Detection and Extraction From Image with Python

[11] ActivityResultContract API

[12] HttpURLConnection API

[13] Executor API

[14] Описание алгоритмов выделения штрихов из текста

[15] Реализация алгоритмов выделения штрихов из текста

[16] Von Gioi R. G. et al. LSD: A line segment detector //Image Processing On Line. – 2012. – T. 2. – С. 35-55.

[17] Canny J. A computational approach to edge detection //IEEE Transactions on pattern analysis and machine intelligence. – 1986. – №. 6. – С. 679-698.

[18] Canny J. A computational approach to edge detection //IEEE Transactions on pattern analysis and machine intelligence. – 1986. – №. 6. – С. 679-698.