## 3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СИСТЕМЫ ЛОКАЛИЗАЦИИ И ОКРАСКИ СТЕН НА ИЗОБРАЖЕНИЯХ

* 1. **Локализация стен методами компьютерного зрения без использования нейронных сетей**

В настоящем проекте библиотека *OpenCV* решает задачи обработки изображения, а также поиска и классификации на нём прямолинейных контуров. *OpenCV* имеет ряд полезных методов для решения задачи нахождения контуров.

Обработка входного изображения производится в несколько этапов, каждый из которых, отдаёт результат собственной работы следующему. Принцип конвейерной ленты позволяет добиться наилучшего результата. Гибкость архитектуры позволяет менять местами или добавлять новые методы обработки, тем самым облегчая разработку.

Первый этап представляет первичную обработку изображения и создание маски над ним. Затем следует функция поиска контуров (*findContours*) библиотеки *OpenCV*, что определяет контур объектов на изображении, основываясь на градиентах граничных частей. Основой для первого этапа является изображение, обработанное методом *Canny* библиотеки *OpenCV*. *Canny* – метод обнаружения края элементов на изображении. Является многоэтапным алгоритмом. На первом этапе алгоритма происходит подавление шума методом Гаусса. Так как обнаружение контуров чувствительно к шуму. Дальше следует нахождение интенсивности изображения за счёт фильтрации ядром Собеля по вертикали и по горизонтали. Этот же этап включает нахождение градиента. На выходе образуется чёрно-белое контурное изображение – рисунок 3.1.

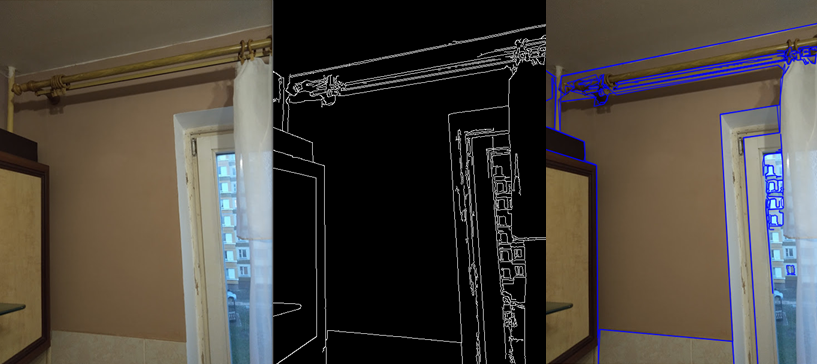


Рисунок 3.1 – Обработка изображения методом *Canny*

Второй этап подразумевает работу непосредственно с самими контурами. На данном этапе происходит отсеивание небольших объектов путём вычисления площадей их контуров. Здесь среднее арифметическое показывает преимущество над медианным средним за счёт достаточного числа небольших шумовых объектов на изображении.

Третий этап работает непосредственно с найденными контурами. На данном этапе идёт сокращение ключевых точек контуров и последующая аппроксимация. Контур приобретает чёткую форму, лишаясь незначительных искривлений (рисунок 3.2).

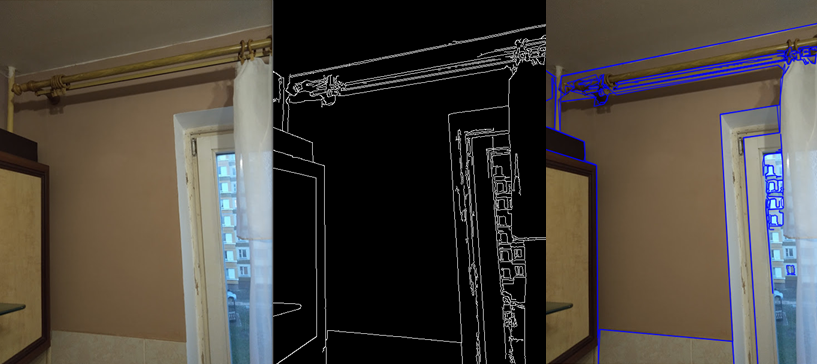


Рисунок 3.2 – Контур изображения после аппроксимации

Четвёртый этап включает функции сортировки контуров на вертикальные и горизонтальные, с учётом отклонения от нормали, а также нахождения точек пересечения двух типов контурных линий. На данном этапе используется функция преобразования Хафа. Преобразование Хафа – вычислительный алгоритм, применяемый для параметрической идентификации геометрических элементов растрового изображения. Чтобы применить преобразование, сначала желательна предварительная обработка края. Для преобразований Хафа линии выражаются в полярной системе координат. В общем случае линию можно обнаружить, определив количество пересечений между кривыми. Чем больше пересекающихся кривых, тем больше точек на линии, представленной этим пересечением. Общий случай даёт возможность определить порог минимального количества пересечений, необходимых для обнаружения линии. Результат представлен на рисунке 3.3.



Рисунок 3.3 – Изображение после обработки методом Хафа

Найденное очертание области графически отображаются на исходном изображении и выводятся пользователю.

Программное обеспечение выполняет задачу распознавания плоскостей на изображениях. Система создана с использованием технологии компьютерного зрения. Распознавание плоскостей на фотографиях является первым этапом в построении системы обнаружения и замены части изображения, в данном случае стены. Потенциально данный инструмент способен облегчить работу дизайнеров интерьера при подборе цветовой гаммы.

Как результат – математические методы без использования нейронных сетей хорошо справляются с точностью определения контуров искомых плоскостей, но не способны отличить стены от других поверхностей, что является существенным недостатком.

* 1. **Локализация стен обученной нейронной сетью**

Для работы с нейронной сетью использовалась библиотека *PyTorch.* За основу взяты модели архитектуры нейронных сетей *MobileNet* и *ResNet,* реализовано переключение между ними.

Скрипт работы с нейронной сетью разбит на несколько секций.

Первая секция – загрузка предварительно сохранённых кодера и декодера. Они и являются основными составляющими обученной нейронной сети и содержат в себе весовые значения.

Далее следует секция объединения кодера и декодера в сегментационный модуль а также переключение модели в режим тестирования методом *eval*.

Следующая часть скрипта посвящена предварительной обработке изображения:

– изменение до размера трёхсот десяти пикселей по большей стороне;

– конвертация в формат *RGB* (*Red Green Blue*);

– нормализация с помощью параметров *std* и *mean* (рекомендуемые документацией *Pytorch* параметры, представляющие из себя массивы чисел);

– преобразование в массив и создание тензора на основе данного массива.

После происходит вызов нейронной сети с флагом no\_*grad* – отключение градиентов.

Завершающая секция производит обратное изменение размера сегментированной маски до изначального размера входного изображения.

Вариант с изменением исходного изображения, прежде чем отправить его в нейронную сеть, использован для увеличения скорости обработки. Обратное увеличение полученной маски до исходного размера приводит к пикселизации контура, как видно из изображения 3.4, но данный нюанс решается постобработкой.



Рисунок 3.4 – Контур сегментационной маски

На рисунке 3.5 представлены этапы обработки изображения: сегментирование изображения нейронной сетью, постобработка полученной маски.

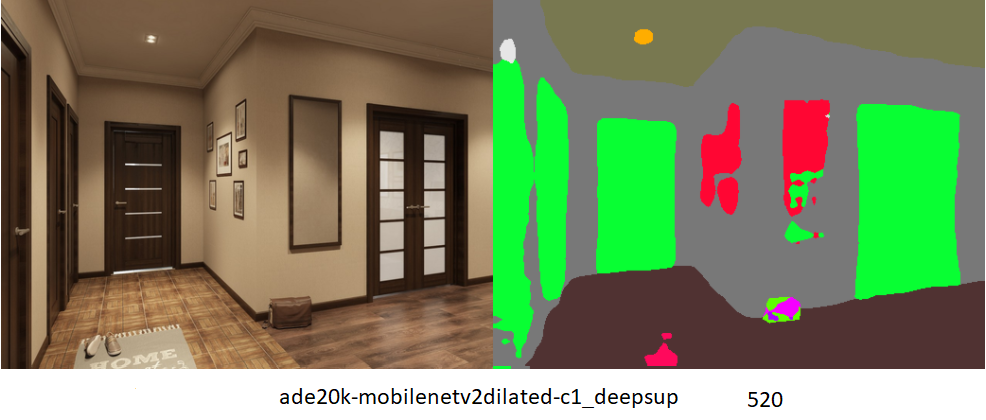


Рисунок 3.5 – Этапы обработки изображения алгоритмом с нейронной сетью

Постобработка результирующего изображения состоит в выделении на сегментированном изображении только маски стены (в примере на изображении 3.5 она представлена серым цветом). Затем маске присваивается выбранный пользователем цвет или текстура, дальше на небольшой процент увеличивается прозрачность. После чего массив представляющий маску объединяется с массивом, представляющим исходное изображение. Перечисленные операции выполняются с помощью библиотеки *numpy*.

Для достижения эстетически приемлемого результата выходной маске также нужна пост обработка в виде сглаживания контуров. Данная проблема также решаема методами библиотеки компьютерного зрения *OpenCv* и библиотеки *numpy*.

* 1. **Структура и алгоритмы на стороне серверного программного обеспечения**

Для применения результатов сегментации кадров в реальных условиях было создано два приложения: сервер и мобильный клиент.

Серверное приложение разворачивается на персональном компьютере пользователя – ПК. Для автоматической загрузки необходимых библиотек и удобной сборки проекта на *java* использовался инструмент *maven*. Файл *pom.xml* содержит настройки сборки проекта и объявления, необходимых для работы проекта, библиотек.

Сервер принимает поступающие от клиента изображения, пробрасывает их через нейронную сеть и возвращает исходный результат клиенту.

Оба приложения используют язык программирования *java*.

После принятия файла с клиентского приложения сервер передаёт его на обработку *python* скрипту, работающему с нейронной сетью либо *python* скрипту, работающему с алгоритмами компьютерного зрения, не использующими нейронную сеть. Полученное результирующее изображение отправляется обратно клиенту.

Использование скрипта на языке *python*, как компонента, для решения основной задачи сервера связано с лёгкостью внесения дополнительных изменений в код, без необходимости перезапускать или перекомпилировать сервер.

Серверное приложение содержит:

–класс сетевого взаимодействия протокола *TCP/IP* (*Server*);

– класс работы с изображениями, в том числе их чтением и записью (*ImgHelper*);

– обученные модели кодера и декодера, нейронных сетей *mobilenet* и *resnet*50, решающих задачу локализации стен на изображениях (*decoder\_epoch\_*20*.pth, encoder\_epoch\_*20*.pth*);

– скрипт на языке *Python* для взаимодействия с нейронными сетями;

– скрипт на языке *Python*, содержащий алгоритмы компьютерного зрения без использования нейронных сетей;

– класс взаимодействия со скриптами на языке *Python* (*PyHelper*);

– класс запуска серверного приложения (*Main*).

Класс *Main*  содержит одноимённый метод и вызывает метод *Start* класса *Server*.

Класс *ImgHelper* содержит методы работы с изображениями, такие как *SaveImg* и *ReadImg*, которые отвечают за сохранение и считывание файла изображения из дискового пространства.

Для работы с *python*-скриптом был создан класс *PyHelper*, имеющий единственный метод *CallPy*. Метод работает с классами *Process* и *Runtime*.

В классе *Server* находятся методы, обрабатывающие входящие подключения от двух типов клиентов. Элементы, находящиеся в классе *Server* можно увидеть в таблице 3.1.

Таблица 3.1 – Элементы класса *Server*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя | Вид элемента | Тип | Модификатор доступа | Краткое описание |
| *SERVER\_PORT* | Переменная | *static final int* | *public* | Хранит значение порта подключения |
| *BUFER\_SIZE* | Переменная | *static final int* | *private* | Хранит значение размера буфера |
| *WorkWithFirstClientType* | Метод | *void* | *private* | Работает с типом клиента, сегментирующим изображения |
| *WorkWithSecondClientType* | Метод | *Void* | *private* | Работает с типом клиента, сегментирующим кадры видео ряда |
| *Start* | Метод | *void* | *public* | Обрабатывает входящие подключения |

В методе *Start* сервер запускает свою работу, начинает обрабатывать входящие подключения от клиентов до тех пор, пока не будет вручную остановлен.

После запуска сервер становится в позицию ожидания подключений. Приложение поддерживает многопользовательский режим, за счёт выделения отдельного потока работы для каждого пользователя. После принятия подключения клиента сервер выделяет ему отдельный поток, в котором обрабатывает все его запросы, а в основном потоке продолжает ожидать следующие подключения.

Сервер имеет консольное отображение информации, преимущественно используемое для отладки и тестирования.

Для подключения клиента к серверу необходимо указать *IP*-адрес (*Internet Protocol Address*) и порт. Порт задан постоянной переменной и не изменим при каждом новом запуске.

Существует несколько вариантов узнать *IP*-адрес ПК. Одним из представительных решений может оказаться широковещательная рассылка. Более простой способ – использовать статический *IP*, который можно узнать, вписав команду *ipconfig* в консоль (*Windows*).

После установки связи клиент обязан сообщить свой тип серверу. Первый тип клиента отвечает за сегментацию изображений и фотографий, второй – за сегментацию кадров видео ряда. Каждый клиент обрабатывается в новом потоке, что позволяет серверу обрабатывать запросы сразу нескольких клиентских приложений.

Для передачи информации по сети используется класс *ServerSocket* и его методы *Accept, GetInputStream, GetOutputStream* и *Close*. Для обработки входящих данных типа *byte* а также их отправки обратно на клиент используются классы *DataInputStream* и *DataOutputStream* соответственно.Вышеупомянутые классы являются наследниками классов *BufferedInputStream* *BufferedOutputStream*, а те в свою очередь, наследуются от *InputStream* и *OutputStream*. Каждое поколение классов имеет своё преимущество. Задачи, решаемые разрабатываемым приложением, не требуют низкоуровнего вмешательства в работу сетевого взаимодействия, благодаря чему можно использовать уже готовые методы для работы со строками, числами и массивами данных типа *byte*.

Класс *Server* имеет два метода для обработки работы двух типов клиентов: *WorkWithFirstClientType* и *WorkWithSecondClientType*.

Подробная работа алгоритма метода *WorkWithFirstClientType* описана ниже.

Работа с изображениями осуществляется посредством класса *BufferedImage*. Объект класса *BufferedImage* сериализуется в массив байт для отправки клиенту осуществляется с помощью класса *ImageIO* и его метода *Write*, который принимает *ByteArrayOutputStream* и записывает в него изображение, представленное *BufferedImage.* Затем объект *ByteArrayOutputStream* преобразовывается в массив байт.

При передаче изображения клиент, сперва отправляет размерность передаваемого массива байт, а затем, отправляет сам массив, который на стороне сервера принимается порционно.

Существует несколько подходов реализации отправки крупных данных по сети. Первый подразумевает одно подключение для одной передачи. То есть, как только файл, изображение, видео или другой документ были переданы, соединение закрывается. Этот способ надёжен, так как при разрыве соединения серверу, считывающему данные, приходит значение-ключ, обозначающее разрыв соединения, который является и сигналом к завершению передачи файла. Минусом данного подхода является неэкономное использование ресурсов. Второй подход – использовать суррогатный ключ завершения передачи, без разрыва самой связи. Минусом данного способа является сложность составления достаточно надёжного ключа, который бы не совпал с передаваемой частью данных. А также необходимость постоянного отслеживания появления данного ключа от принимающей стороны.

Ещё одним способом является первичная передача данных о размере последующего передаваемого файла принимающей стороне. Плюсом данного способа является относительно несложная реализация и отсутствие разрыва соединения.

Последний способ наиболее подходит для реализации передачи изображений или кадров клиентом. Так как разрывать подключение между передачей первого и второго кадра видео ряда не удачное решение в виду экономии времени пользователя и ресурсов памяти.

Метод принимает размер будущего изображения, далее происходит считывание входящего потока и запись их порциями в *OutputStream* до тех пор, пока размер считанного массива не достигнет заявленного размера клиентом. Затем *OutputStream* с помощью метода *ToByteArray* переводится в массив байт.

Изображения передаются пакетами размерностью с буфер конкретного приложения, как на серверном так и на клиентском программном обеспечении (далее ПО) значение буфера выставлено в 1024\*2. Число выбрано как компромисс между скоростью передачи данных и вероятностью их потери.

Несмотря на размер буфера, размер реально передаваемых данных может отличаться как в большую, так и в меньшую сторону. Данное поведение обусловлено возможными помехами в сетевом взаимодействии и нестабильностью сетевого сигнала в целом.

Вышеупомянутое поведение выставляет ограничение на использование точного подсчёта порций передаваемых данных. Но данная проблема решается используемым механизмом передачи больших файлов.

* 1. **Структура и алгоритмы на стороне клиентского программного обеспечения**

Мобильное приложение подразумевает несколько вариантов использования: сегментирование отдельных изображений, сегментирование кадров с камеры в реальном режиме времени. В зависимости от выбранного варианта, сервер будет обрабатывать одни и те же запросы по-разному.

Платформа разработки мобильного клиента – *Android*, как одна из самых распространённых мобильных операционных систем. Язык программирования – *java*.

Клиентское приложение содержит:

–класс сетевого взаимодействия протокола *TCP/IP* (*MobSocket*);

–классы двух основных экранов приложения (*MainActivity* и *WorkActivity*);

– классы фрагментов приложения (*VideoFragment* и *PictureFragment*);

– класс работы с *Camera* *API* (*MobCameraAPI*);

– ресурсы разметки пользовательского интерфейса (*activity\_main.xml*, *activity\_work.xml*, *fragment\_picture.xml* и *fragment\_video.xml*).

Приложение имеет два основных окна: окно подключения (*MainActivity*), которое состоит из двух полей для ввода параметров подключения и кнопки «*connect*», и рабочее окно (*WorkActivity*), которое становится доступным после успешного подключения к серверу.

В таблице 3.2 отображены элементы класса *MainActivity* разрабатываемого мобильного приложения.

Таблица 3.2 – Элементы класса *MainActivity*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя | Вид элемента | Тип | Модификатор доступа | Краткое описание |
| *b\_enter* | Переменная | *Button* | *private* | Кнопка устанавливающая соединение |
| *port* | Переменная | *EditText* | *private* | Поле ввода порта для подключения |
| *ip* | Переменная | *EditText* | *private* | Поле ввода IP адреса для подключения |
| *onCreate* | Метод | *void* | *protected* | Обрабатывает событие нажатия на кнопку установки соединения |
| *MobTask* | Вложенный класс | *AsyncTask* | *private* | Специализированный класс асинхронных задач |

Как только пользователь ввёл все необходимые данные для подключения и нажал соответствующую кнопку, создаётся новый экземпляр асинхронного класса *MobTask*, который является наследником базового класса *AsynkTask*. *MobTask* является вложенным классом *MainActivity,* и не доступен для кода из вне.

В классе *MobTask* создаётся сокет подключения, принимается сообщение от сервера, подтверждающее успешность подключения и возвращается результат операции в виде булевого значения.

Выполнять масштабные операции в основном потоке, работающем с пользовательским интерфейсом, не является хорошей практикой. Операции, такие как: работа с сетью или с базой данных, могут отнимать достаточно большое количество времени. Выполнение подобных операция в основном потоке заблокирует на время их исполнения отклик пользовательского интерфейса. Со стороны пользователя данное явление выглядит как зависание программы. Это может негативно сказаться на пользовательском опыте. Избежать подобной ситуации можно используя потоки или асинхронное программирование.

Существует ограничение – параллельным потокам запрещён доступ к элементам управления. И если от результата исполнения тяжёлой операции зависит представление пользовательского интерфейса, уместней использовать класс, разработанный специально для подобных нужд – *AsynkTask*. Он является абстрактным и требует собственной реализации от разработчика.

Основными методами класса *AsynkTask* можно назвать: *doInBackground* (в нём необходимо реализовать код тяжёлой операции), *onPostExecute* и *onPreExecute* (методы, исполняющиеся до и после операции, в основном потоке, то есть имеющие доступ к интерфейсу пользователя). Конструктор *AsynkTask* принимает на вход неограниченное число параметров заданного в объявлении класса типа. Также можно задать типы параметров, возвращаемых при завершении операции или при промежуточных выводах.

Выше описанные составляющие доказывают удобство использования методов данного класса для решения задач с сетевым соединением.

Во время подключения в *MainActivity* создается сокет подключения, позволяющий обмениваться данными между устройствами. Класс, отвечающий за сетевое соединение – *MobSoket*.

Задача передачи сокета между *Activity* решается паттерном *SingleTone*. Такое решение заменяет глобальную переменную, а также ограничивает создание новых экземпляров класса без необходимости.

Класс *MobSoket*, помимо реализации *SingleTone*, имеет шесть основных рабочих методов, не считая перегрузки: *connectToDevice*, *unconnectToDevice*, *sendMessage* , *getMessage SendFile* и *GetFile*.

В методе *connectToDevice* вызывается метод Connect у объекта сокета. Метод *unconnectToDevice* закрывает все открытые соединения, в том числе и те, что были использованы для передачи или получения данных.

Метод *sendMessage* создаёт или использует объект класса *DataOutputStream*, используемый для отправки сообщения на сервер. И непосредственно отправляет сообщение на сервер.

Метод *getMessage* создаёт или использует объект класса *DataInputStream*, используемый для получения сообщения от сервера в строковом формате.

Также есть два подобных вышеописанным метода *SendFile* и *GetFile*. Они предназначены для отправки объёмных сообщений, как правило, файлов, а в контексте настоящего приложения – изображений. Методы работают с массивом байт и содержат логику характерную для отправки или принятия объёмного объекта.

После успешного подключения к серверу происходит переход на экран *WorkActivity*, который располагает тремя кнопками: «*unconnect*», «*picture*» и «realtime\_video».

Кнопка «*unconnect*» отправляет на сервер числовое значение ноль, что является признаком завершения сеанса с пользователем. Следующим действием разрывает соединение, вызывая метод *Unconnect* класса *MobSocket*, который закрывает выходящий и исходящий потоки, а затем и сам сокет. Клиентское же приложение снова переходит в окно подключения. Подобное мягкое решение разрыва связи создано для обеспечения возможности серверу корректно завершить работу с данным клиентом и обработкой его данных.

Первым вариантом использования мобильного клиента является – передача изображений. Импортировать изображение можно выбрав из галереи настоящего устройства или сделав снимок. Клиент предназначен для разового анализа. Удобен в обработке изображений, сделанных в прошлом, и не требователен к качеству связи и техническим характеристикам мобильного устройства.

Второй тип клиента отвечает за передачу кадров видео, а именно трансляцию последнего на сервер в реальном времени. Ответ от сервера поступает также в реальном времени и выводится на экран приложения.

Кнопки «*picture*» и «realtime\_video» отвечают за тип клиента, при их нажатии происходит отправка на сервер сообщения с номером типа клиента, а также создаётся и отображается соответствующий фрагмент. Для определения типа фрагмента использован фабричный метод, возвращающий необходимый класс, по числу. Для выполнения операций по отправки сообщений на сервер создан вложенный класс *ClientTypeTask*, наследующий *AsynkTask*. Он принимает входным значением число, идентифицирующее тип клиента. Отправляет его значение на сервер и, в случае успешной отправки, создаёт и открывает соответствующий фрагмент.

Приложение имеет два класса фрагментов: *FragmentPicture* и *FragmentVideo*.

*FragmentPicture* содержит кнопки, отвечающие за импорт и отправку на сервер изображений: «*upload photo*», «*send photo*». А также объект *ImageView*, отвечающий за отображение исходного или результирующего изображения на экране мобильного приложения для пользователя. В таблице 3.3 описаны основные компоненты класса *FragmentPicture.*

Таблица 3.3 – Основные элементы класса *FragmentPicture*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя | Вид элемента | Тип | Модификатор доступа | Краткое описание |
| *mobSocket* | Переменная | *MobSocket* | *private* | Класс сетевого взаимодействия |
| *ControllerTask* | Вложенный класс | *AsyncTask* | *private* | Специализированный класс асинхронных задач |

Продолжение таблицы 3.3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *selectImage* | Метод | *void* | *private* | Контекстное меню импорта изображений |
| *onActivityResult* | Метод | *void* | *public* | Логика обработки импортированного изображения |
| *onCreateView* | Метод | *View* | *public* | Содержит логику обработки нажатий кнопок, расположенных во фрагменте |

Класс *FragmentPicture* содержит вложенный асинхронный класс – *ControllerTask*. Его задачей является асинхронный вызов метода отправки файла, класса *MobSocket*. при нажатии управляющей кнопки, создаётся новый объект класса ControllerTask.

Сервер работает с изображениями формата *jpeg* или *jpg* (название на старых платформах, не поддерживающих формат файла больше трёх символов). Соответственно при импорте из галереи файл конвертируется в необходимый формат перед преобразованием в массив байт. При создании фото – изображения имеют необходимый формат по умолчанию.

После нажатия на кнопку импорт изображения перед пользователем появляется диалоговое окно со списком отзывчивых элементов: *Take Photo, Choose from Gallery, Cancel.*

После выбора изображение передаётся элементу отображения на экране.

Обработанное изображение снова отображается в элементе *ImageView*.

Класс *FragmentVideo* схож по функционалу с классом *FragmentPicture*, за исключением использования дополнительного класса *MobCameraAPI* для захвата кадров с камеры в реальном времени.