МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

РАЗРАБОТКА ПРИЛОЖЕНИЙ И DIRECTSHOW-ФИЛЬТРА ДЛЯ ПЕРЕДАЧИ ВИДЕОПОТОКА С КАМЕРЫ МОБИЛЬНОГО УСТРОЙСТВА В ПРИЛОЖЕНИЯ НА WINDOWS ЧЕРЕЗ WI-FI

КУРСОВАЯ РАБОТА

студента 3 курса 351 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Голышева Юрия Олеговича

Научный руководитель
доцент, к. ф.-м. н. _____ А. С. Иванова

Заведующий кафедрой
доцент, к. ф.-м. н. _____ С. В. Миронов

Саратов 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В последние годы наблюдается стремительный рост популярности видеосвязи и онлайн-трансляций, что обусловило повышенный спрос на качественные и доступные решения для передачи видеопотока между различными устройствами. Одной из актуальных задач является использование камеры мобильного телефона в качестве веб-камеры для персонального компьютера, что позволяет существенно повысить качество изображения по сравнению с большинством недорогих USB-камер. Или же получить возможность использовать видеосвязь с персонального компьютера, если камеры нет вовсе.

Существующие программные решения, такие как DroidCam, и аналогичные, предоставляют подобную функциональность, однако зачастую имеют ограничения по качеству, стабильности работы, совместимости с различными приложениями или требуют покупки. Кроме того, многие из них используют закрытые протоколы передачи данных, что затрудняет их интеграцию в собственные проекты или расширение функциональности.

Задачи:

1. Разработать мобильное приложение для захвата и передачи видеопотока с камеры Android-устройства.
2. Реализовать серверное приложение для Windows, принимающее и отображающее видеопоток, а также передающее его в виртуальную камеру.
3. Создать виртуальную камеру на базе DirectShow для интеграции видеопотока в сторонние приложения.
4. Обеспечить корректное взаимодействие всех компонентов системы и провести тестирование полученного решения.

1 Средства разработки и языки программирования

Для реализации поставленной задачи были выбраны современные и широко используемые инструменты и языки программирования, что позволило обеспечить высокую производительность, кроссплатформенность отдельных компонентов и удобство дальнейшей поддержки проекта.

1.1 Мобильное приложение

Для разработки клиентской части, осуществляющей захват и передачу видеопотока с камеры мобильного устройства, был выбран язык Kotlin и среда разработки Android Studio. Kotlin обеспечивает лаконичный и безопасный синтаксис, а Android Studio предоставляет мощные средства для отладки, профилирования и тестирования мобильных приложений. Для работы с камерой использовалась библиотека CameraX, что позволило реализовать современный и гибкий захват видеопотока.

1.2 Серверное приложение

Для создания серверной части, принимающей видеопоток, отображающей его на пользовательском интерфейсе и взаимодействующей с виртуальной камерой, был выбран язык C# и платформа .NET (WPF). C# и WPF позволяют быстро создавать современные графические интерфейсы, а также обеспечивают удобную работу с сетевыми протоколами, потоками и обработкой изображений. Средой разработки выступила Visual Studio, предоставляющая широкий набор инструментов для отладки и профилирования приложений под Windows.

1.3 Виртуальная камера

Для реализации виртуальной камеры был выбран язык C++ и технология DirectShow. DirectShow — это стандартная мультимедийная платформа Windows для работы с потоками аудио и видео, а C++ позволяет создавать высокопроизводительные драйверы и фильтры, интегрируемые на уровне системы. Разработка велась в Visual Studio с использованием соответствующих SDK и библиотек Windows.

2 Реализация мобильного приложения

Мобильное приложение для Android является отправной точкой всей системы и отвечает за захват видеопотока с камеры устройства, его предварительную обработку и передачу данных на серверное приложение, работающее на компьютере с Windows. Ключевыми требованиями к мобильному клиенту стали высокая производительность, стабильная работа при длительном использовании. В данной главе подробно рассматриваются архитектурные решения, используемые технологии и особенности реализации мобильного приложения, а также приводятся ключевые фрагменты кода, иллюстрирующие работу основных модулей.

2.1 Захват и обработка видеопотока

Захват и предварительная обработка видеопотока в мобильном приложении реализованы с использованием библиотеки CameraX и механизма анализа изображений ImageAnalysis.

CameraX — это современная библиотека от Google для работы с камерой на Android, которая значительно упрощает интеграцию камеры в приложения, обеспечивает совместимость с разными устройствами и предоставляет удобный API для управления параметрами съёмки (разрешение, частота кадров, выбор камеры и т.д.).

ImageAnalysis — это специальный компонент CameraX, позволяющий получать доступ к "сырым"кадрам с камеры в реальном времени для их последующей обработки или передачи. Такой подход позволяет получать изображения непосредственно с камеры устройства с минимальной задержкой и гибко управлять параметрами съёмки.

Для захвата видеопотока создаётся объект ImageAnalysis, который настраивается на работу с максимальным разрешением (до 1920×1080) и частотой кадров до 30 FPS. В качестве формата выходных данных выбран YUV_420_888 — это стандартный формат для большинства камер Android, обеспечивающий оптимальное соотношение качества и производительности.

```
1 val imageAnalysis = ImageAnalysis.Builder()
2     .setResolutionSelector(resolutionSelector)
3     .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
4     .setOutputImageFormat(ImageAnalysis.OUTPUT_IMAGE_FORMAT_YUV_420_888)
5     .build()
```

Для обработки кадров используется отдельный поток-исполнитель `cameraExecutor`, что позволяет не блокировать основной UI-поток и обрабатывать кадры максимально быстро.

Для предотвращения потерь кадров и обеспечения стабильной работы используется очередь кадров (ArrayBlockingQueue<ImageProxy>), в которую помещаются новые кадры по мере их поступления. Если очередь заполнена, новые кадры отбрасываются, чтобы не создавать задержек и не перегружать систему.

```
val frameQueue: ArrayBlockingQueue<ImageProxy> = ArrayBlockingQueue(3)
```

Обработка и сжатие кадров выполняются в пуле потоков (encodePool), что позволяет параллельно обрабатывать несколько кадров и эффективно использовать ресурсы устройства.

Каждый кадр, поступающий в очередь, проходит следующие этапы обработки:

1. Преобразование из YUV в NV21:
   Функция copyImageToNV21Optimized преобразует кадр из формата YUV_420_888 (который возвращает CameraX) в формат NV21. Конвертируем кадр в формат NV21, потому что этот формат поддерживается стандартным классом YuvImage в Android, который позволяет быстро и эффективно сжимать изображение в JPEG. Формат NV21 широко используется для работы с камерой и обеспечивает совместимость с большинством инструментов обработки изображений на Android, а также позволяет минимизировать задержки при преобразовании и сжатии кадров для передачи по сети.
   Вот как copyImageToNV21Optimized работает по шагам:
   а) Получение размеров и буферов:
   Из объекта ImageProxy извлекаются ширина и высота изображения. Затем получаются три плоскости (planes): Y (яркость), U и V (цветность).
   б) Получение stride-ов:
   Для каждой плоскости определяется шаг по строке (rowStride) и шаг по пикселю (pixelStride), что важно для правильного копирования данных из буферов.
   в) Копирование Y-плоскости (яркость):

Если шаг по строке (yRowStride) совпадает с шириной изображения, все данные яркости можно скопировать одним блоком. Если нет — копирование происходит построчно: из каждой строки буфера Y копируется только нужное количество байт (ширина изображения).

г) Копирование UV-плоскости (цветность):

Рассчитываются размеры хрома-плоскости (ширина и высота вдвое меньше, чем у Y). Если данные UV уже хранятся в памяти в формате, совместимом с NV21 (т.е. uvPixelStride == 2 и uvRowStride == width), то копирование происходит максимально быстро: из каждой строки поочерёдно берутся байты из V и U и записываются в итоговый массив. Если формат не совпадает (например, на некоторых устройствах), используется универсальный (но более медленный) способ: для каждого пикселя отдельно вычисляются индексы в буферах U и V, и поочерёдно копируются байты в итоговый массив.

д) Результат:

На выходе получается массив байт в формате NV21, где сначала идут все значения яркости (Y), а затем чередующиеся значения V и U для каждого пикселя хрома.

Функция максимально эффективно преобразует кадр из формата, возвращаемого камерой, в формат NV21, используя быстрые пути копирования там, где это возможно, и универсальный — для всех остальных случаев. Это позволяет быстро и без лишних аллокаций подготовить данные для последующего сжатия в JPEG.

2. Сжатие в JPEG:

Полученный массив NV21 передаётся в объект YuvImage, который сжимает изображение в формат JPEG с заданным качеством (например, 40%).

```
1        val yuvImage = YuvImage(reusableNV21, ImageFormat.NV21, width,
         ↪  height, null)
2        yuvImage.compressToJpeg(Rect(0, 0, width, height), 40,
         ↪  reusableBaos)
3        val jpegData = reusableBaos.toByteArray()
```

Оптимизация производительности:

1. Использование очереди и пула потоков позволяет обрабатывать кадры параллельно, минимизируя задержки и обеспечивая стабильную частоту кадров.

2. Для снижения нагрузки на сборщик мусора и ускорения работы используются переиспользуемые массивы байт reusableNV21 (NV21 кадр) и reusableBaos (JPEG кадр).
3. В процессе работы ведётся подробное логирование времени обработки, сжатия и отправки каждого кадра, а также статистика по FPS, размеру очереди и количеству отброшенных кадров.

В результате каждого цикла обработки формируется компактный JPEG-кадр, который готов к передаче по сети на серверное приложение. Такой подход позволяет достичь высокой производительности (около 30 кадров в секунду при Full HD) и минимальной задержки, что критически важно для использования видеопотока в реальном времени.

2.2 Сетевое взаимодействие и отправка данных

После захвата и сжатия каждого кадра в формате JPEG мобильное приложение осуществляет передачу данных на серверное приложение, работающее на компьютере с Windows. Для этого используется прямое TCP-соединение по Wi-Fi, что обеспечивает минимальную задержку и высокую надёжность передачи. Перед началом передачи пользователь указывает IP-адрес и порт сервера (по умолчанию 8888) в интерфейсе приложения. При нажатии кнопки «Старт» приложение инициирует подключение к серверу с помощью стандартного TCP-сокета:

```
val socket = Socket(ipAddress, port).apply {
    soTimeout = 5000
    keepAlive = true
    tcpNoDelay = true
    sendBufferSize = 2_097_152
    receiveBufferSize = 1_048_576
}
val stream = socket.getOutputStream()
```

Параметры сокета подобраны для максимальной производительности: отключён алгоритм Нейгла (tcpNoDelay), увеличены буферы отправки и приёма, установлен таймаут.

Для передачи кадров используется простой, но надёжный бинарный протокол, который позволяет серверу однозначно выделять границы каждого кадра и восстанавливать поток даже при ошибках сети. Каждый кадр инкапсулируется следующим образом:

— FRAME_START_MARKER — уникальная сигнатура начала кадра (4 байта).
— Размер кадра — 4 байта (Little Endian), указывающие длину JPEG-данных.
— JPEG-данные — непосредственно сжатое изображение.
— FRAME_END_MARKER — уникальная сигнатура конца кадра (4 байта).

Передача каждого кадра осуществляется в отдельном потоке с синхронизацией, чтобы избежать конфликтов при одновременной обработке нескольких кадров. Для каждого кадра формируется пакет и отправляется по следующей схеме:

```
val header = ByteBuffer.allocate(4).putInt(finalJpeg.size).array()
stream.write(FRAME_START_MARKER)
stream.write(header)
stream.write(finalJpeg)
stream.write(FRAME_END_MARKER)
stream.flush()
```

После вызова flush() все данные, накопленные в буфере потока при вызовах write(), немедленно отправляются по сети на сервер. Отправка происходит в бинарном виде.

Весь процесс защищён блокировкой (socketSendLock), чтобы несколько потоков не отправляли данные одновременно и не нарушали структуру потока. В случае возникновения ошибок соединения (например, разрыв Wi-Fi, недоступность сервера) приложение корректно завершает передачу, уведомляет пользователя о проблеме и позволяет повторно инициировать подключение. Также реализована обработка исключений при отправке данных, что предотвращает зависание приложения.

В процессе передачи ведётся сбор статистики: количество отправленных кадров, среднее время обработки и передачи, количество отброшенных кадров. Эта информация используется для логирования и отображения статуса пользователю, а также для оптимизации работы приложения.

Такой подход к сетевому взаимодействию обеспечивает надёжную и быструю передачу видеопотока с минимальными задержками, что особенно важно для использования камеры в режиме реального времени.

2.3 Особенности реализации и оптимизации

В процессе разработки мобильного приложения особое внимание уделялось вопросам производительности, стабильности работы и минимизации задержек при передаче видеопотока. Для этого были реализованы ряд технических решений и оптимизаций, позволяющих эффективно использовать ресурсы устройства и обеспечивать высокое качество передачи данных.

Для предотвращения "узких мест"и потерь кадров в приложении реализована архитектура с использованием нескольких потоков:

- Основной поток камеры занимается только захватом изображений и помещает их в ограниченную очередь (ArrayBlockingQueue<ImageProxy>).
- Пул потоков обработки (обычно 2 потока) параллельно извлекает кадры из очереди, выполняет их преобразование и сжатие, а затем отправляет по сети.
- Такой подход позволяет избежать блокировок и обеспечивает стабильную частоту кадров даже при высоких нагрузках.

Для снижения нагрузки на сборщик мусора и ускорения работы используются переиспользуемые массивы байт для хранения промежуточных данных (например, буфер NV21 и ByteArrayOutputStream для JPEG). Это позволяет минимизировать количество аллокаций памяти и избежать зависаний при работе с большими изображениями.

В коде предусмотрены проверки и логирование поддерживаемых камерой разрешений и диапазонов FPS. Это позволяет быстро диагностировать возможные проблемы совместимости.

Для предотвращения перехода устройства в спящий режим во время работы приложения используется механизм WakeLock, который гарантирует, что процесс захвата и передачи кадров не будет прерван системой.

```
1 val pm = getSystemService(Context.POWER_SERVICE) as PowerManager
2 wakeLock = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
  ↪ "CameraStream::Wakelock")
3 wakeLock.acquire()
```

В приложении реализована обработка всех возможных ошибок на каждом этапе: от захвата кадра до отправки по сети. В случае возникновения исключений приложение корректно завершает текущий поток передачи, уведомляет пользователя и позволяет быстро восстановить работу.

В процессе работы ведётся подробное логирование времени обработки каждого кадра, времени сжатия, времени передачи по сети, а также статистика по FPS, размеру очереди и количеству отброшенных кадров. Это позволяет выявлять и устранять узкие места, а также оптимизировать работу приложения под реальные условия эксплуатации.

В коде предусмотрена возможность гибко настраивать параметры сжатия JPEG (качество), разрешение и частоту кадров, что позволяет адаптировать приложение под разные сценарии использования — от максимального качества до минимальной задержки.

Благодаря этим решениям мобильное приложение демонстрирует высокую производительность, стабильную работу и минимальные задержки передачи видеопотока, что критически важно для использования камеры телефона в качестве веб-камеры в реальном времени.

2.4 Использование QR кода для быстрого подключения

Для повышения удобства пользователя в приложении реализована функция сканирования QR-кода, позволяющая автоматически заполнить поля IP-адреса и порта сервера. Это особенно актуально при первом подключении или при смене сети, когда ручной ввод параметров может быть неудобен и подвержен ошибкам.

В интерфейсе приложения предусмотрена отдельная кнопка «QR», при нажатии на которую запускается сканер QR-кода на основе библиотеки ZXing. После успешного сканирования приложение автоматически извлекает из QR-кода необходимые параметры подключения (IP и порт) и подставляет их в соответствующие поля, что позволяет начать передачу видеопотока буквально в пару кликов.

```
1 val qrLauncher = rememberLauncherForActivityResult(
2     contract = ActivityResultContracts.StartActivityForResult()
3 ) { res ->
4     val result: IntentResult? =
      ↪  IntentIntegrator.parseActivityResult(res.resultCode, res.data)
5     if (result != null && result.contents != null) {
6         val uri = Uri.parse(result.contents)
7         if (uri.scheme == "cam") {
8             ipAddress = uri.host ?: ""
9             if (uri.port != -1) port = uri.port.toString()
```

```
10              Toast.makeText(context, "Считано $ipAddress",
                ↪ Toast.LENGTH_SHORT).show()
11          }
12      }
13  }
```

На стороне серверного приложения (Windows) предусмотрена генерация QR-кода с нужными параметрами, который пользователь может отсканировать камерой телефона.

Такой подход значительно ускоряет процесс настройки соединения, снижает вероятность ошибок и делает использование системы максимально простым даже для неподготовленных пользователей.

3 Реализация серверного приложения для Windows

Серверное приложение для Windows выполняет центральную роль в системе: оно принимает видеопоток, поступающий с мобильного устройства, обеспечивает его декодирование и отображение на пользовательском интерфейсе, а также передаёт кадры в виртуальную камеру для дальнейшего использования в сторонних программах. Приложение реализовано на языке C# с использованием платформы .NET (WPF), что позволило создать современный и удобный интерфейс, а также обеспечить высокую производительность и надёжность работы.

В данной главе подробно рассматриваются архитектура, ключевые модули и особенности реализации серверного приложения, а также приводятся примеры кода, иллюстрирующие работу основных компонентов.

3.1 Сетевой сервер и приём кадров

Одной из важнейших задач серверного приложения является организация надёжного и производительного приёма видеопотока, поступающего с мобильного устройства по сети. Для этого в приложении реализован собственный TCP-сервер, который слушает указанный порт (по умолчанию 8888) и принимает входящие соединения от клиентов.

После установления соединения сервер ожидает поступления данных в соответствии с заранее согласованным протоколом: каждый кадр инкапсулируется в специальную структуру с маркерами начала и конца, а также содержит информацию о размере JPEG-данных. Такой подход позволяет однозначно выделять границы кадров, корректно обрабатывать разрывы соединения и восстанавливать поток даже при ошибках передачи.

Весь процесс приёма кадров реализован асинхронно, что позволяет одновременно обслуживать несколько клиентов и не блокировать основной поток интерфейса. Принятые кадры сразу же декодируются и отображаются на UI, а также сохраняются для последующей передачи в виртуальную камеру.

Пример кода приёма и разбора кадра:

```
1 // Ожидание маркера начала кадра
2 var startBuf = await ReadExactAsync(stream, FRAME_START_MARKER.Length,
  ↪  cancellationToken);
3 if (!startBuf.AsSpan().SequenceEqual(FRAME_START_MARKER)) continue;
4
5 // Чтение размера кадра
```

```
var sizeBuf = await ReadExactAsync(stream, 4, cancellationToken);
int frameSize = BitConverter.ToInt32(sizeBuf, 0);

// Получение JPEG-данных
byte[] frameData = await ReadExactAsync(stream, frameSize, cancellationToken);

// Проверка маркера конца кадра
var endBuf = await ReadExactAsync(stream, FRAME_END_MARKER.Length,
↪  cancellationToken);
```

функция ReadExactAsync предназначена для гарантированного чтения точного количества байтов из сетевого потока (NetworkStream), даже если данные поступают частями.

```
private static async Task<byte[]> ReadExactAsync(NetworkStream stream, int
↪  size, CancellationToken token)
        {
            byte[] buf = new byte[size];
            int pos = 0;
            while (pos < size)
            {
                int read = await stream.ReadAsync(buf, pos, size - pos,
                ↪  token);
                if (read == 0)
                    throw new IOException("Соединение разорвано");
                pos += read;
            }
            return buf;
        }
```

Такой подход обеспечивает надёжную и быструю обработку видеопотока, минимизирует задержки и позволяет поддерживать высокую частоту кадров даже при интенсивной нагрузке.

3.2 Отображение видеопотока на UI

После успешного приёма и разбора каждого кадра серверное приложение выполняет его декодирование и отображение на пользовательском интерфейсе. Это позволяет пользователю в реальном времени видеть изображение, поступающее с камеры мобильного устройства, а также контролировать качество и стабильность передачи.

Каждый кадр, полученный от клиента, представляет собой массив байт в формате JPEG. Для его отображения на UI используется следующий подход:

— Сначала JPEG-данные преобразуются в объект BitmapImage с помощью стандартных средств .NET.
— Для ускорения работы и предотвращения блокировки интерфейса декодирование выполняется в отдельном потоке, а обновление UI — через диспетчер интерфейса (Dispatcher).

Пример кода декодирования и отображения кадра можно посмотреть в приложении Б.

Для предотвращения избыточной нагрузки на интерфейс и обеспечения плавности отображения реализовано ограничение частоты обновления UI (обычно до 60 кадров в секунду). Это достигается с помощью проверки временного интервала между обновлениями и пропуска лишних кадров, если они поступают слишком часто.

На интерфейсе приложения также отображается дополнительная информация:

— Текущий статус соединения и передачи данных.
— Размер и разрешение последнего кадра.
— Частота кадров (FPS) и скорость передачи данных (КБ/с).

Это позволяет пользователю оперативно оценивать качество работы системы и при необходимости быстро реагировать на возможные проблемы.

Такой подход обеспечивает не только наглядность и удобство использования приложения, но и высокую производительность даже при передаче видеопотока в высоком разрешении и с большой частотой кадров.

3.3 Интеграция с виртуальной камерой

Одной из ключевых функций серверного приложения является возможность передачи полученного видеопотока в виртуальную камеру, которая становится доступной для любых программ на компьютере (Max, OBS, Telegram и др.) как обычное устройство видеозахвата. Для этого реализована интеграция с драйвером виртуальной камеры через механизм разделяемой памяти (shared memory).

Сама виртуальная камера регистрируется и удаляется из системы по нажатию на кнопки «Зарегистрировать виртуальную камеру» и «Удалить виртуальную камеру» соответственно. Виртуальная камера вшита при сборке в выходной

исполняемый файл.

Взаимодействие с виртуальной камерой организовано с помощью специального класса VirtualCameraManager, который инкапсулирует всю логику работы с разделяемой памятью и обеспечивает удобный интерфейс для передачи кадров. Внутри используется вспомогательный клиент VirtualCameraSharedMemClient, который напрямую работает с памятью, доступной как серверному приложению, так и драйверу виртуальной камеры (DirectShow-фильтру).

Перед началом передачи кадров приложение инициализирует соединение с виртуальной камерой и открывает разделяемую память:

```
var success = await _virtualCameraManager.InitializeAsync("Android Cam");
if (success)
{
    await _virtualCameraManager.StartAsync();
}
```

В процессе инициализации создаётся или открывается объект памяти с фиксированным именем (Global \\vCamShm) и размером, достаточным для хранения двух кадров в формате BGR24 (Full HD), а также служебной информации (frameId, dataSize, currentBuffer).

Каждый раз при получении нового кадра (после декодирования JPEG) приложение масштабирует изображение до нужного разрешения (1920×1080), преобразует его в формат BGR24 и записывает в активный буфер разделяемой памяти. Для этого вызывается асинхронный метод:

```
await _virtualCameraManager.SendFrameAsync(imageData);
```

Внутри происходит:

- Декодирование JPEG-данных в Bitmap.
- Масштабирование до Full HD.
- Преобразование в массив байт BGR24.
- Запись данных в активный буфер разделяемой памяти.
- Обновление служебных полей (frameId, dataSize, currentBuffer) для синхронизации с драйвером виртуальной камеры.

Для предотвращения конфликтов при одновременном доступе к памяти используется двойной буфер и атомарные переменные. Это позволяет виртуальной камере всегда получать целостный кадр, даже если в этот момент происходит запись нового изображения.

Серверное приложение также предоставляет пользователю возможность вручную регистрировать или удалять виртуальную камеру в системе, а также отслеживать её статус и возможные ошибки через события и сообщения интерфейса.

Такой подход обеспечивает быструю и надёжную передачу видеопотока в виртуальную камеру, минимизирует задержки и гарантирует совместимость с большинством программ, работающих с веб-камерами на Windows.

3.4 Безопасность и удобство

В процессе разработки серверного приложения особое внимание уделялось вопросам безопасности работы в локальной сети, а также удобству для конечного пользователя. Были реализованы следующие механизмы и функции:

— Для корректной работы приложения и возможности приёма видеопотока с мобильного устройства по сети необходимо, чтобы выбранный порт (по умолчанию 8888) был открыт для входящих соединений. Серверное приложение автоматически проверяет наличие соответствующего правила в брандмауэре Windows и, при необходимости, создаёт его с помощью встроенных средств операционной системы. Это избавляет пользователя от необходимости вручную настраивать сетевые параметры и снижает вероятность ошибок при первом запуске.

— Перед запуском сервера приложение проверяет, не занят ли выбранный порт другим процессом. В случае обнаружения конфликта пользователь получает подробное сообщение с рекомендациями по устранению проблемы (например, закрыть другое приложение, изменить порт или перезагрузить компьютер). Это позволяет избежать неочевидных сбоев и облегчает диагностику сетевых проблем.

— Всё сетевое взаимодействие и обработка кадров реализованы асинхронно, что позволяет не блокировать интерфейс и обеспечивает высокую отзывчивость приложения даже при возникновении ошибок или задержек в сети. В случае возникновения исключительных ситуаций (разрыв соединения, повреждённые данные, ошибки декодирования) приложение корректно завершает обработку, уведомляет пользователя и позволяет быстро восстановить работу без необходимости перезапуска.

Пользовательский интерфейс приложения разработан с учётом принципов простоты и наглядности:

— Вся основная информация о статусе соединения, частоте кадров, размере и разрешении текущего изображения отображается в реальном времени.
— Для быстрого подключения реализована поддержка сканирования QR-кода, что позволяет избежать ручного ввода IP-адреса и порта.
— Управление виртуальной камерой (регистрация, удаление, статус) осуществляется в пару кликов.

В приложении реализовано подробное логирование всех ключевых событий и ошибок, что облегчает диагностику и поддержку системы. Пользователь всегда получает понятные сообщения о текущем состоянии приложения и возможных проблемах.

3.5 Быстрое подключение с помощью QR-кода

Для генерации QR-кода в десктоп-приложении используется современная библиотека QRCoder, которая интегрируется в проект на .NET и позволяет создавать QR-коды непосредственно в памяти приложения. После запуска сервера и определения IP-адреса компьютера программа формирует специальную строку подключения в формате cam://<ip-адрес>:<порт>, например, cam://192.168.1.100:8888. Эта строка кодируется в QR-код, который отображается в отдельном окне или прямо в основном интерфейсе программы.

Пользователь может воспользоваться функцией сканирования QR-кода в мобильном приложении. После сканирования параметры подключения автоматически извлекаются из кода и подставляются в соответствующие поля, что позволяет начать передачу видеопотока буквально в один клик. Такой подход полностью исключает необходимость ручного ввода IP-адреса и порта, минимизирует риск ошибок и значительно ускоряет процесс первого запуска.

Использование QR-кода делает процесс подключения мобильного приложения к серверу максимально простым и интуитивно понятным даже для неподготовленных пользователей. Это решение не только ускоряет настройку, но и делает систему более современной и удобной в повседневном использовании.

4 Реализация виртуальной камеры

Виртуальная камера завершает цепочку передачи видеопотока и делает изображение, поступающее с мобильного устройства, доступным для любых программ Windows, поддерживающих работу с веб-камерами. Для этого был разработан собственный DirectShow-фильтр на языке C++, который взаимодействует с серверным приложением через разделяемую память и предоставляет видеопоток в стандартных форматах (YUY2, NV12, RGB24 и др.).

В этой главе рассматриваются архитектура, ключевые механизмы и особенности реализации виртуальной камеры, обеспечивающие высокую производительность, совместимость и надёжность работы.

4.1 Архитектура виртуальной камеры

Виртуальная камера реализована как DirectShow-фильтр, зарегистрированный в системе как стандартное устройство видеозахвата. Это позволяет использовать её в любых приложениях (Discord, OBS, Telegram и др.) без дополнительной настройки.

Ключевые элементы архитектуры:

— Разделяемая память (shared memory):
Для передачи кадров между серверным приложением и драйвером виртуальной камеры используется именованная разделяемая память с фиксированным именем (Global \\vCamShm). В этой памяти размещается структура SharedHeader, содержащая два буфера для кадров (двойной буфер), а также служебные поля: атомарный идентификатор кадра (frameId), размер данных (dataSize) и индекс активного буфера (currentBuffer).

— Двойной буфер и атомарные переменные:
Такой подход позволяет избежать состояния гонки: пока серверное приложение записывает новый кадр в один буфер, фильтр виртуальной камеры читает предыдущий из другого. Переключение буферов и обновление служебных полей происходит атомарно, что гарантирует целостность данных и минимальные задержки.

— Преобразование форматов:
Виртуальная камера хранит кадры в формате BGR24 (Full HD), но при запросе от приложений может конвертировать их в требуемый формат (YUY2, NV12, RGB24 и др.) и масштабировать изображение, если это

необходимо.

— Доступ к памяти:

  Для доступа к разделяемой памяти используется класс-обёртка SharedMem (Код можно посмотреть в приложении В), который открывает и отображает память в адресное пространство процесса фильтра. При каждом запросе кадра фильтр проверяет актуальность данных, копирует их из нужного буфера и выполняет необходимое преобразование.

— Регистрация и совместимость:

  После установки фильтр регистрируется в системе как устройство видеозахвата, что обеспечивает его видимость для большинства программ Windows.

Преимущества такой архитектуры:

— Высокая производительность и минимальные задержки благодаря прямому обмену данными через память.

— Гарантия целостности кадров и отсутствие конфликтов при одновременном доступе.

— Гибкость в поддержке различных видеоформатов и разрешений.

— Простота интеграции с любыми приложениями, ожидающими стандартную веб-камеру.

Эта архитектура позволяет реализовать надёжную и быструю виртуальную камеру, полностью совместимую с экосистемой Windows и современными требованиями к качеству видеопотока.

4.2 Получение и обработка кадров

Виртуальная камера получает видеокадры из разделяемой памяти, которую заполняет серверное приложение. Этот процесс реализован с учётом высокой производительности, синхронизации и необходимости поддерживать различные форматы вывода для совместимости с программами Windows.

Внутри фильтра создаётся объект класса SharedMem, который открывает именованную разделяемую память (Global \\vCamShm) и предоставляет доступ к структуре SharedHeader. Эта структура содержит два буфера для кадров, а также атомарные переменные:

— frameId — уникальный идентификатор текущего кадра,

— dataSize — размер данных кадра,

— currentBuffer — индекс активного буфера (0 или 1).

При каждом запросе нового кадра от приложения (например, Discord или OBS) фильтр:

1. Проверяет, изменился ли frameId по сравнению с предыдущим запросом. Если да — это новый кадр, и его нужно обработать.
2. Определяет, какой из двух буферов (currentBuffer) содержит актуальные данные.
3. Получает указатель на массив байт с изображением в формате BGR24 (Full HD).

В зависимости от того, какой формат запрашивает приложение, фильтр выполняет соответствующее преобразование:

— YUY2:

Каждый блок из двух пикселей BGR преобразуется в четыре байта YUY2 с помощью формул. При необходимости выполняется масштабирование по ширине и высоте.

— NV12:

Из BGR24 формируется плоскость яркости (Y) и объединённая плоскость цветности (UV), также с возможным масштабированием.

— RGB24:

Если запрошен исходный формат и разрешение совпадает, данные просто копируются. Если разрешение отличается — выполняется обрезка или масштабирование.

— I420:

Аналогично NV12, но с раздельными плоскостями U и V. Весь процесс реализован максимально эффективно: используются прямые указатели, минимальное количество копирований, а масштабирование выполняется по формуле пересчёта координат.

Использование двойного буфера и атомарных переменных гарантирует, что фильтр всегда получает целостный кадр, даже если в этот момент серверное приложение записывает новый. Это исключает появление разрывов или частично обновлённых изображений.

Такой подход обеспечивает высокую производительность, минимальные задержки и совместимость с большинством программ, ожидающих стандартные форматы видеопотока от веб-камеры.

4.3 Регистрация и взаимодействие с приложениями

Виртуальная камера, реализованная в виде DirectShow-фильтра, интегрируется в операционную систему Windows как стандартное устройство видеозахвата. Это позволяет использовать её в любых приложениях, поддерживающих работу с веб-камерами, без необходимости дополнительной настройки или установки стороннего ПО.

После сборки и установки драйвера-фильтра выполняется его регистрация в системе. Для этого используется стандартный механизм регистрации COM-объектов и фильтров DirectShow. В процессе регистрации фильтр получает уникальный идентификатор (GUID) и описание, а также указывает поддерживаемые форматы видеопотока (YUY2, NV12, RGB24, I420) и и разрешения (например, 1920×1080, 1280×720 и др.).

В результате фильтр становится видимым для всех приложений, которые используют стандартные средства Windows для поиска и работы с видеоустройствами.

Когда стороннее приложение (например, Discord, OBS, Skype, Telegram и др.) запрашивает список доступных видеоустройств, виртуальная камера появляется в этом списке наряду с физическими веб-камерами. При выборе виртуальной камеры приложение начинает запрашивать видеокадры через стандартный интерфейс DirectShow.

Виртуальная камера автоматически подстраивается под формат запрошенный приложением(YUY2, NV12, RGB24, I420).

При каждом запросе кадра фильтр получает актуальные данные из разделяемой памяти, преобразует их в нужный формат и возвращает приложению. Благодаря поддержке нескольких форматов обеспечивается максимальная совместимость с различными программами.

Таким образом, виртуальная камера становится универсальным и удобным инструментом для передачи видеопотока с мобильного устройства в любые приложения Windows.

5 Взаимодействие компонентов системы

Вся система построена по модульному принципу, где каждый компонент выполняет свою функцию, а их взаимодействие обеспечивает передачу видеопотока от камеры мобильного устройства до конечного приложения на компьютере. В этой главе подробно рассматривается, как происходит обмен данными между компонентами, какие протоколы и механизмы используются для синхронизации, а также как обеспечивается целостность и минимальная задержка видеопотока.

5.1 Общий сценарий работы

1. Запуск серверного приложения на ПК:
   Пользователь запускает серверное приложение, которое автоматически настраивает брандмауэр, открывает нужный порт и, при необходимости, отображает QR-код с параметрами подключения.
2. Подключение мобильного приложения:
   Пользователь запускает мобильное приложение, сканирует QR-код или вручную вводит IP-адрес и порт сервера, после чего инициирует передачу видеопотока.
3. Передача кадров по сети:
   Мобильное приложение захватывает кадры с камеры, сжимает их в JPEG и отправляет по TCP-соединению на серверное приложение. Для каждого кадра используется собственный протокол с маркерами начала и конца, а также указанием размера данных.
4. Приём и обработка на сервере:
   Серверное приложение асинхронно принимает кадры, декодирует их, отображает на UI и, при необходимости, передаёт в виртуальную камеру через разделяемую память.
5. Передача в виртуальную камеру:
   Кадры масштабируются до нужного разрешения, преобразуются в формат BGR24 и записываются в активный буфер разделяемой памяти. Для синхронизации используется двойной буфер и атомарные переменные.
6. Получение кадра виртуальной камерой:
   Драйвер виртуальной камеры (DirectShow-фильтр) читает актуальный кадр из разделяемой памяти, преобразует его в нужный формат (YUY2, NV12,

RGB24 и др.) и отдаёт стороннему приложению.

7. Использование в сторонних приложениях:

   Любое приложение (Discord, OBS, Skype и др.) может выбрать виртуальную камеру как источник видео и получать с неё поток в реальном времени.

5.2 Протоколы и механизмы синхронизации

Для передачи кадров между мобильным и серверным приложением используется TCP-соединение и простой бинарный протокол с маркерами и размером данных. Это обеспечивает надёжность и позволяет корректно восстанавливать поток даже при ошибках передачи.

Для обмена кадрами между серверным приложением и виртуальной камерой используется именованная shared memory. Двойной буфер и атомарные переменные (frameId, dataSize, currentBuffer) гарантируют целостность данных и отсутствие состояния гонки.

Все операции по приёму, обработке и передаче кадров реализованы асинхронно, что позволяет минимизировать задержки и не блокировать пользовательский интерфейс.

Таким образом, взаимодействие компонентов системы построено так, чтобы обеспечить надёжную, быструю и удобную передачу видеопотока с мобильного устройства в любые приложения Windows, требующие веб-камеру.

6 Тестирование

Для оценки работоспособности и производительности разработанной системы было проведено комплексное тестирование всех её компонентов в реальных условиях. Тестирование включало проверку передачи видеопотока с мобильного устройства на ПК, отображения видео в серверном приложении, а также интеграции с виртуальной камерой и сторонними программами.

6.1 Запуск и подключение

В ходе тестирования серверное приложение было запущено на компьютере с Windows, после чего был сгенерирован QR-код с параметрами подключения. Мобильное приложение успешно считало QR-код, автоматически заполнив поля IP-адреса и порта, что позволило быстро и без ошибок начать передачу видеопотока.

[Скриншот 1: Окно серверного приложения с отображением QR-кода и статусом ожидания подключения] [Скриншот 2: Интерфейс мобильного приложения с заполненными полями IP и порта после сканирования QR-кода]

6.2 Передача и отображение видеопотока

После подключения мобильного приложения видеопоток с камеры телефона начал поступать на серверное приложение. На пользовательском интерфейсе ПК кадры отображались в реальном времени с частотой около 30 кадров в секунду и разрешением Full HD (1920×1080). Задержка между захватом кадра на телефоне и его появлением на экране ПК была минимальной и практически незаметной.

[Скриншот 3: Серверное приложение с отображаемым видеопотоком и информацией о FPS, разрешении и размере кадра]

6.3 Интеграция с виртуальной камерой

Включение виртуальной камеры позволило сделать видеопоток доступным для любых сторонних программ. Виртуальная камера корректно определялась в системе как стандартное устройство видеозахвата. В таких приложениях, как Discord, OBS Studio и Telegram, можно было выбрать виртуальную камеру в качестве источника видео и получать изображение с мобильного устройства в реальном времени.

[Скриншот 4: Настройки видео в Discord с выбранной виртуальной камерой и отображением видеопотока] [Скриншот 5: Окно OBS Studio с добавленной виртуальной камерой в качестве источника]

6.4 Оценка производительности и стабильности

В процессе тестирования система стабильно работала на протяжении длительного времени, не наблюдалось потерь кадров или существенных задержек. Частота кадров держалась на уровне 30 FPS, а качество изображения соответствовало Full HD.

[Скриншот 6: Серверное приложение с отображением статистики FPS и размера кадра при длительной работе]

6.5 Совместимость

Система была протестирована с различными популярными программами для видеосвязи и стриминга. Везде виртуальная камера определялась корректно, а видеопоток отображался без искажений и с минимальной задержкой. [Скриншот 7: Пример работы виртуальной камеры в Telegram или Skype]

6.6 Результаты тестирования

Результаты тестирования подтвердили работоспособность и эффективность разработанной системы. Все компоненты взаимодействуют корректно, обеспечивая передачу видеопотока с мобильного устройства на ПК в режиме реального времени с высоким качеством и стабильностью.

7 Итоги и идеи для дальнейшего развития проекта

В ходе выполнения курсовой работы были успешно решены все поставленные задачи. Разработано мобильное приложение для Android, способное захватывать видеопоток с камеры устройства, сжимать кадры и передавать их по сети на компьютер. Создано серверное приложение для Windows, которое принимает видеопоток, отображает его на пользовательском интерфейсе и обеспечивает передачу кадров в виртуальную камеру. Реализован драйвер виртуальной камеры на базе DirectShow, что позволило сделать видеопоток доступным для любых сторонних приложений, поддерживающих работу с веб-камерами.

Все компоненты системы интегрированы между собой, обеспечено их корректное взаимодействие и синхронизация. Проведено тестирование решения, в ходе которого подтверждена его работоспособность, стабильность передачи видеопотока с частотой до 30 кадров в секунду и разрешением Full HD, а также совместимость с популярными программами (Discord, OBS, Skype и др.).

В результате реализованная система позволяет использовать камеру мобильного устройства в качестве полноценной веб-камеры для ПК, что значительно расширяет возможности пользователя и повышает качество видеосвязи.

Идеи для дальнейшего развития проекта:

1. Оптимизация мобильного приложения, снижение нагрузки.
2. Добавление поддержки передачи и обработки аудиопотока с микрофона мобильного устройства.
3. Реализация автоматического обнаружения серверов в локальной сети.
4. Разработка кроссплатформенных версий приложений (например, для macOS и iOS).
5. Улучшение интерфейса пользователя и расширение возможностей управления камерой (выбор камеры, настройки разрешения и частоты кадров, эффекты).
6. Более подробное и понятное уведомление пользователя об ошибках.

ЗАКЛЮЧЕНИЕ

В ходе данной работы:

1. Было разработано мобильное приложение для захвата и передачи видеопотока с камеры Android-устройства.
2. Было Реализовано серверное приложение для Windows, принимающее и отображающее видеопоток, а также передающее его в виртуальную камеру.
3. Была создана виртуальную камеру на базе DirectShow для интеграции видеопотока в сторонние приложения.
4. Было обеспеченно корректное взаимодействие всех компонентов системы и проведено тестирование полученного решения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

ПРИЛОЖЕНИЕ А

Flash-носитель с отчетом о выполненной работе

Папка `Coursework` — LaTeX- вариант курсовой работы.

Папка `Game` — полный проект игры.

`Coursework.pdf` — курсовая работа.

ПРИЛОЖЕНИЕ Б

Код декодирования и отображения кадра

```
_ = Task.Run(() =>
{
    try
    {
        BitmapImage bmp;
        using (var ms = new MemoryStream(frameData))
        {
            bmp = new BitmapImage();
            bmp.BeginInit();
            bmp.CacheOption = BitmapCacheOption.OnLoad;
            bmp.StreamSource = ms;
            bmp.EndInit();
            bmp.Freeze();
        }

        var nowTicks = Stopwatch.GetTimestamp();
        if (nowTicks - _lastUiFrameTicks >= _uiFrameIntervalTicks)
        {
            _lastUiFrameTicks = nowTicks;
            Dispatcher.Invoke(() =>
            {
                CameraImage.Source = bmp;
                StatusText.Text = $"Кадр: {frameSize / 1024} KB";
                ResolutionText.Text = $"{bmp.PixelWidth}×{bmp.PixelHeight}";
            });
        }
    }
    catch (Exception ex)
    {
        Dispatcher.Invoke(() => StatusText.Text = $"Ошибка изображения:
        ↪ {ex.Message}");
    }
});
```

ПРИЛОЖЕНИЕ В

```
1  // Константы Full HD кадра (BGR24)
2  constexpr DWORD FRAME_W   = 1920;
3  constexpr DWORD FRAME_H   = 1080;
4  constexpr DWORD FRAME_BPP = 3;        // BGR24
5  constexpr DWORD FRAME_SZ  = FRAME_W * FRAME_H * FRAME_BPP; // 6 220 800 байт
6
7  // Структура, лежащая в разделяемой памяти
8  struct SharedHeader
9  {
10     //std::atomic<LONG> frameId;   // Атомарный ID кадра // инкремент при
       ↪ каждом новом кадре
11     //std::atomic<DWORD> dataSize; // Атомарный размер данных // должно быть
       ↪ FRAME_SZ
12     //BYTE data[FRAME_SZ];
13     std::atomic<LONG> frameId;
14     std::atomic<DWORD> dataSize;
15     std::atomic<int> currentBuffer; // 0 или 1
16     BYTE data[2][FRAME_SZ];        // Два буфера
17 };
18
19 // Класс-обёртка для доступа (только чтение) к shared memory
20 class SharedMem
21 {
22     HANDLE hMap  = nullptr;
23     SharedHeader* pMem = nullptr;
24     int openAttempts = 0;
25
26 public:
27     bool Open()
28     {
29         if (hMap) return true;
30
31         // Пытаемся несколько раз, с небольшими задержками
32         for (int i = 0; i < 3; i++)
33         {
34             hMap = ::OpenFileMappingW(FILE_MAP_READ, FALSE,
                   ↪ L"Global\\vCamShm");
35             if (hMap) break;
36             ::Sleep(100); // Ждем 100 мс перед следующей попыткой
```

```
37          }
38
39          openAttempts++;
40
41          if (!hMap) return false;
42          pMem = reinterpret_cast<SharedHeader*>(::MapViewOfFile(hMap,
            ↪  FILE_MAP_READ, 0, 0, sizeof(SharedHeader)));
43          return pMem != nullptr;
44      }
45
46      const SharedHeader* Get() const { return pMem; }
47
48      ~SharedMem()
49      {
50          if (pMem) ::UnmapViewOfFile(pMem);
51          if (hMap) ::CloseHandle(hMap);
52      }
53  };
```