

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра дискретной математики и алгоритмики

**Улучшение качества видеопотока с помощью нейронных сетей на
графических процессорах**

Курсовая работа

Бинцаровского Леонида Петровича
студента 3 курса
специальность «информатика»

Научный руководитель:
старший преподаватель
Д. И. Пирштук

Минск, 2024

Реферат

Курсовая работа, 32 стр., 9 иллюстр., 6 источников.

Ключевые слова: MediaPipe; C++; Swift; Metal; XNNPack; инференс.

Объекты исследования — улучшение качества видеопотока с помощью нейронных сетей на графических процессорах.

Цель исследования — реализация конвейера для улучшения качества видеопотока с помощью нейронных сетей на графических процессорах и приложений для тестирования данного конвейера на macOS, Linux и iOS.

Методы исследования — системный подход, изучение соответствующей литературы и электронных источников, постановка задачи и её решение.

В результате исследования были разработаны конвейеры для улучшения качества видеопотока на macOS, Linux и iOS. Реализованы приложения на C++ и Swift для запуска и тестирования данных конвейеров.

Области применения — инференс нейронных сетей.

Оглавление

Введение	3
1 Постановка задачи	4
2 Обзор фреймворка MediaPipe. Основные элементы конвейера	5
2.1 Знакомство с MediaPipe	5
2.2 Основные элементы конвейера	5
2.2.1 Пакет (Packet)	5
2.2.2 Узлы (Nodes or calculator)	5
2.2.3 Поток (Streams)	6
2.2.4 Граф (Graph)	6
2.2.5 Конвейер (Pipeline)	7
3 Разработка конвейера улучшения видеопотока	8
3.1 Разработка архитектуры конвейера	8
3.2 Конвейер для улучшения качества видеопотока на macOS	12
3.3 Конвейер для улучшения качества видеопотока на iOS и Linux	14
4 Реализация приложений для запуска графов	16
4.1 Запуск графа на macOS и Linux	16
4.2 Запуск графа на iOS	20
Заключение	31
Список использованных источников	32

Введение

Современные технологии стремительно развиваются, проникая во все сферы нашей жизни. Одной из ключевых задач становится обработка и анализ видеопотоков, что позволяет улучшать пользовательский опыт, создавать новые форматы взаимодействия и автоматизировать сложные процессы.

В настоящее время обработка видеопотоков и редактирование контента стали неотъемлемой частью современной технологической среды. Одним из направлений в этой области является улучшение качества видеопотока. Есть множество вариантов: увеличение разрешения, стилизация в определённом стиле, удаление шума (*denoising*), улучшение цветопередачи, добавление эффекта глубины, автоматическое центрирование объекта и многое другое.

На данный момент в большинстве сфер ит-направления применяются нейронные сети. И обработка видеопотока не осталась в стороне. Почти все вышеперечисленные варианты для улучшения качества видео могут быть реализованы с помощью нейронных сетей.

В данном контексте фреймворк MediaPipe представляет собой мощный инструмент для решения задач обработки видеопотоков. Разработанный компанией Google, MediaPipe предоставляет широкий набор инструментов и библиотек для анализа и модификации видеоданных в реальном времени с помощью нейронных сетей.

ГЛАВА 1

ПОСТАНОВКА ЗАДАЧИ

На данный момент существует огромное количество моделей предназначенные под всевозможные цели: улучшение качества изображения, перевод изображения из черно-белого в цветное, распознавание речи и перевод её в текст, сегментация объектов, перевод текста на разные языки и т.д. В рамках работы будут использованы модели для стилизации видео whitebox_cartoon_gan [2].



Рисунок 1.1 — Пример использования модели whitebox_cartoon_gan для стилизации фото

Данные модели принимают тензор изображения в формате BGR со значениями в диапазоне $[-1.0; 1.0]$. Для реализации под macOS будет использована модель с входными данными 360×640 . На платформах Linux и iOS будет задействована модель с входными данными 640×360 из-за особенности расположения дисплея на iOS.

Итак, сформулируем задачу улучшения качества видеопотока с помощью нейронных сетей на графических процессорах, которая будет изучаться в данной работе. Необходимо разработать конвейеры для инференса выбранных моделей под платформы macOS, Linux и iOS. На вход будет предоставлен кадр с камеры устройства, на выходе - получен стилизованный кадр. Для тестирования работы конвейеров будут реализованы приложения на языках C++ и Swift под соответствующие платформы.

ГЛАВА 2

ОБЗОР ФРЕЙМВОРКА MEDIAPIPE. ОСНОВНЫЕ ЭЛЕМЕНТЫ КОНВЕЙЕРА

2.1 Знакомство с MediaPipe

MediaPipe — один из самых обширных фреймворков для запуска конвейеров (предобработка данных, запуск (inference) модели, постобработка результатов модели) машинного обучения, позволяющий упростить написание кроссплатформенного кода для запуска предобученных моделей. Эта структура может использоваться для различных приложений для обработки изображений и мультимедиа (особенно в виртуальной реальности), таких как обнаружение объектов, распознавание лиц, отслеживание рук, отслеживание нескольких рук и сегментация волос. MediaPipe поддерживает различные аппаратные и операционные платформы, такие как Android, iOS и Linux, предлагая API на C++, Java, Objective-c и т.д.

В контексте поставленной задачи будет использован фреймворк MediaPipe на языке программирования C++ и Swift для платформ macOS, Linux и iOS.

2.2 Основные элементы конвейера

2.2.1 Пакет (Packet)

Пакет (англ. Packet) — единица данных, перемещаемая по потокам и обрабатываемая калькулятором. Каждый пакет несёт в себе данные определённого типа — это может быть строка, целое число, массив чисел с плавающей запятой или пользовательский тип, описанный и сериализуемый в protobuf. Каждый пакет содержит в себе timestamp — отметку времени, ассоциированную с пакетом. Она напрямую не связана с реальным временем, так как нужна для того, чтобы отличать, какой пакет был раньше, какой позже.

2.2.2 Узлы (Nodes or calculator)

Узлы (англ. Nodes) создают и/или обрабатывают Packet, и именно на них приходится основная часть работы графа. По историческим причинам их также называют calculator. У каждого калькулятора должен быть как минимум один входящий и как минимум один исходящий поток. Калькулятор представляет из себя C++ класс, реализующий интерфейс CalculatorBase:

- `static ::mediapipe::Status GetContract(CalculatorContract*);`
статический метод, в котором калькулятор описывает форматы данных, которые ждет на вход и готов отдать на выход;
- `::mediapipe::Status Open(CalculatorContext*);`
инициализация калькулятора при создании графа. Здесь, например, может быть загрузка данных, требуемых для работы;
- `::mediapipe::Status Process(CalculatorContext*);`
обработка поступившего пакета;
- `::mediapipe::Status Close(CalculatorContext*);`
закрытие вершины.

2.2.3 Поток (Streams)

Ребра графа (англ. Streams) задают связи между калькуляторами. С помощью потоков по графу перемещаются пакеты с данными. Поток может быть внутренним, входным (input) и исходящим (output). Внутренний поток соединяет два калькулятора, по входному потоку из внешнего кода в граф попадают данные, а с помощью исходящего потока граф отправляет данные наружу, в вызывающий код.

2.2.4 Граф (Graph)

Обработка данных в MediaPipe происходит внутри графа (англ. Graph), который определяет пути потока пакетов между узлами. Граф может иметь любое количество входов и выходов, а поток данных может разветвляться и сливаться. Как правило, данные идут вперед, но возможны и обратные циклы.



Рисунок 2.1 —
Простейший граф
(визуализация)

```

1  input_stream: "in"
2  output_stream: "out"
3
4  node {
5      calculator: "RepeatNTimesCalculator"
6      input_stream: "in"
7      output_stream: "OUTPUT_TAG:out"
8      node_options: {
9          [RepeatNTimesCalculatorOptions] {
10             n: 3
11         }
12     }
13 }

```

Рисунок 2.2 — Простейший граф
(pbtxt)

2.2.5 Конвейер (Pipeline)

Конвейер в MediaPipe задается в форме графа. Графы описываются в формате protobuf text file (pbtxt). MediaPipe позволяет из калькуляторов составлять необходимый конвейер для запуска модели, а затем просто встраивать его в приложения на разных платформах. Сейчас разработчики заявляют о поддержке нескольких дистрибутивов Linux, WSL, MacOS, Android, iOS. В MediaPipe есть встроенные калькуляторы для запуска TensorFlow и TFLite моделей.

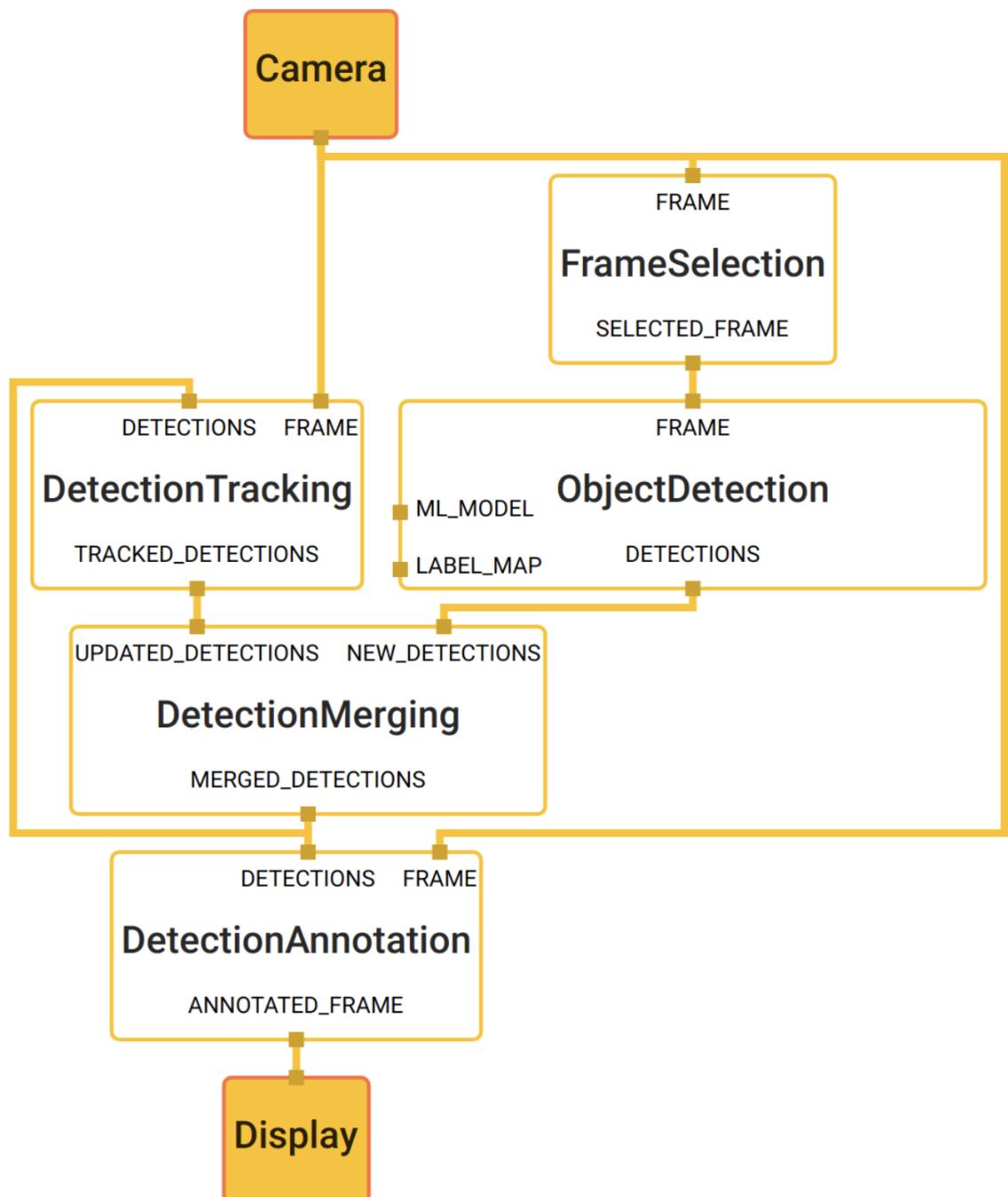


Рисунок 2.3 – Пример конвейера

ГЛАВА 3

РАЗРАБОТКА КОНВЕЙЕРА УЛУЧШЕНИЯ ВИДЕОПОТОКА

3.1 Разработка архитектуры конвейера

Для запуска модели необходимо создать конвейер, который в случае с фреймворком MediaPipe представляет собой граф из некоторого количества калькуляторов. В этом пункте определим архитектуру общего конвейера, а в следующих внесем корректизы для определённых платформ и сетей.

На вход в конвейер будет поступать текущее изображение видеопотока. На выходе конвейер будет возвращать текущий кадр с примененными улучшениями. В зависимости от платформы тип входного изображения будет изменяться. Так на Linux и iOS в конвейер будет поступать кадр типа mediapipe::GpuBuffer, на macOS же будет поступать кадр типа mediapipe::ImageFrame.

```
1 # CPU/GPU buffer containing the current frame. (type: ImageFrame/GpuBuffer)
2 input_stream: "input_video"
3
4 # Output image with rendered results. (type: ImageFrame/GpuBuffer)
5 output_stream: "output_video"
```

Листинг 3.1: Инициализация входных пакетов в графе конвейера

Для предотвращения скопления узлов в очередь на получение изображений и данных (это приводило бы к увеличению задержек и расходу памяти) используется калькулятор FlowLimiterCalculator. Кроме того, данный калькулятор устраняет ненужные вычисления, например, вывод, произведенный узлом, может быть передан далее по потоку, если последующие узлы все еще заняты обработкой предыдущих входных данных. Для управления потоком калькулятор пропускает первое входящее изображение без изменений и ждет, пока нижележащие узлы графа (калькуляторы или подграфы) не закончат свои вычисления, прежде чем пропустить еще одно изображение.

На вход данный калькулятор принимает текущий кадр видеопотока и поток FINISHED от указанного калькулятора конвейера. Так как последний калькулятор конвейера будет лишь переводить из одного формата в другой и это не будет занимать много времени, необходимо ждать выполнения калькулятора преобразования тензора в изображение. В input_stream_info становится параметр, который указывает, в каком состоянии должен быть поток FINISHED, для того чтобы калькулятор пропустил следующее изображение. Если поток FINISHED будет не пустым, калькулятор передаст следующее изображение, иначе они будут отбрасываться.

```
1 node {
```

```

2   calculator: "FlowLimiterCalculator"
3   input_stream: "input_video"
4   input_stream: "FINISHED:output_image"
5   input_stream_info: {
6     tag_index: "FINISHED"
7     back_edge: true
8   }
9   output_stream: "throttled_input_video"
10 }

```

Листинг 3.2: Вызов калькулятора FlowLimiterCalculator в графе конвейера

После получения отфильтрованного кадра в пакет throttled_input_video нужно провести необходимую обработку для получения подходящего на вход модели тензора. Для этого используется калькулятор ImageToTensorCalculator. Данный калькулятор является универсальным и даёт возможность получить входной тензор для кадра любого типа (ImageFrame и GpuBuffer). Это возможно, так как этот калькулятор реализован на Metal (для iOS), OpenGL (для Linux), OpenCV (для macOS).

На вход калькулятор ImageToTensorCalculator принимает текущий кадр из FlowLimiterCalculator. Возвращает готовый для инференса тензор. Во время превращения изображение подготавливается в подходящий для модели формат. Для этого указываются дополнительные параметры:

- **output_tensor_width** ширина ожидаемого на вход модели изображения;
- **output_tensor_height** высота ожидаемого на вход модели изображения;
- **keep_aspect_ratio** устанавливает нужно ли при переходе ко входным размерам модели сохранять соотношение сторон как у исходного изображения;
- **output_tensor_float_range** указывает минимальное и максимальное ожидаемые значения типа float;
- **gpu_origin** указывает, как расположено изображение. Для OpenGL входное изображение начинается снизу и его нужно перевернуть по вертикали, так как тензоры должны начинаться сверху;
- **border_mode** указывает метод пиксельной экстраполяции. При преобразовании изображения в тензор может произойти, что тензору потребуется считать пиксели за пределами границ изображения. Этот параметр указывает, как такие пиксели будут вычисляться.

```

1 node {
2   calculator: "ImageToTensorCalculator"
3   input_stream: "IMAGE:throttled_input_video"

```

```

4     output_stream: "TENSORS:image_tensor"
5     options: {
6         [mediapipe.ImageToTensorCalculatorOptions.ext] {
7             output_tensor_width: 'width'
8             output_tensor_height: 'height'
9             keep_aspect_ratio: false
10            output_tensor_float_range {
11                min: 'min_float'
12                max: 'max_float'
13            }
14            gpu_origin: 'CONVENTIONAL'
15            border_mode: BORDER_ZERO
16        }
17    }
18}

```

Листинг 3.3: Вызов калькулятора ImageToTensorCalculator в графе конвейера

Далее полученный тензор отправляется на инференс модели. Для этого используется калькулятор InferenceCalculator. На вход данный калькулятор принимает тензор входного изображения в подходящем размере. Возвращает данный калькулятор тензор обработанного изображения. Дополнительно указывается путь до модели, которая будет использована в данном графе, и делегат, который будет отвечать за запуск модели. Этот калькулятор можно считать универсальным и не зависящим от платформы, так как он поддерживает реализацию таких делегаторов, как XNNPack (для любого CPU, но в этом случае для macOS) и GPU (Metal для iOS и OpenGL для Linux).

```

1 node {
2     calculator: "InferenceCalculator"
3     input_stream: "TENSORS:image_tensor"
4     output_stream: "TENSORS:denoised_tensor"
5     options: {
6         [mediapipe.InferenceCalculatorOptions.ext] {
7             model_path: "path/to/the/model"
8             use_gpu: 'true/false'
9             use_nnapi: 'true/false'
10            cpu_num_thread: '-1'
11            delegate { 'delegate' {} }
12        }
13    }
14}

```

Листинг 3.4: Вызов калькулятора InferenceCalculator в графе конвейера

На следующем шаге тензор, полученный от InferenceCalculator, восстанавливается в выходное изображение. Для этого используется калькулятор TensorToImageCalculator. На вход он принимает тензор для восстановления. Возвращает изображение формата mediapipe::Image. Для корректного восстановления изображения из тензора необходимо указать дополнительные параметры:

- **gpu_origin** указывает, как расположено изображение. Для OpenGL входное изображение начинается снизу и его нужно перевернуть по вертикали, так как тензоры должны начинаться сверху;

- **input_tensor_float_range:min** минимальное ожидаемое значение типа float;
- **input_tensor_float_range:max** максимальное ожидаемое значение типа float;
- **tensor_position** указывает какой входной тензор нарезать, если в модели несколько выходных тензоров.

Данный калькулятор также является универсальным, так как реализован на Metal (для iOS), OpenGL (для Linux) и OpenCV (для macOS).

```

1 node {
2     calculator: "TensorsToImageCalculator"
3     input_stream: "TENSORS: denoised_tensor"
4     output_stream: "IMAGE: output_image"
5     options: {
6         [mediapipe.TensorsToImageCalculatorOptions.ext] {
7             gpu_origin: "CONVENTIONAL"
8             input_tensor_float_range {
9                 min: 'min'
10                max: 'max'
11            }
12            tensor_position: 0
13        }
14    }
15 }
```

Листинг 3.5: Вызов калькулятора TensorToImageCalculator в графе конвейера

Для получения итогового изображения осталось перевести, полученное в калькуляторе TensorToImageCalculator, из формата mediapipe::Image в mediapipe::ImageFrame для CPU на macOS и в mediapipe::GpuBuffer для GPU на iOS и Linux. Для этого используется калькулятор FromImageCalculator.

```

1 node {
2     calculator: "FromImageCalculator"
3     input_stream: "IMAGE: output_image"
4     output_stream: "IMAGE_CPU: output_video"
5     output_stream: "IMAGE_GPU: output_video"
6 }
```

Листинг 3.6: Вызов калькулятора FromImageCalculator в графе конвейера

На данном калькуляторе описание общей архитектуры конвейера заканчивается. Для лучшего понимания полноценной архитектуры прилагается визуализированный график конвейера (рис 4.1). Далее рассмотрим конкретные графы для macOS, iOS и Linux.

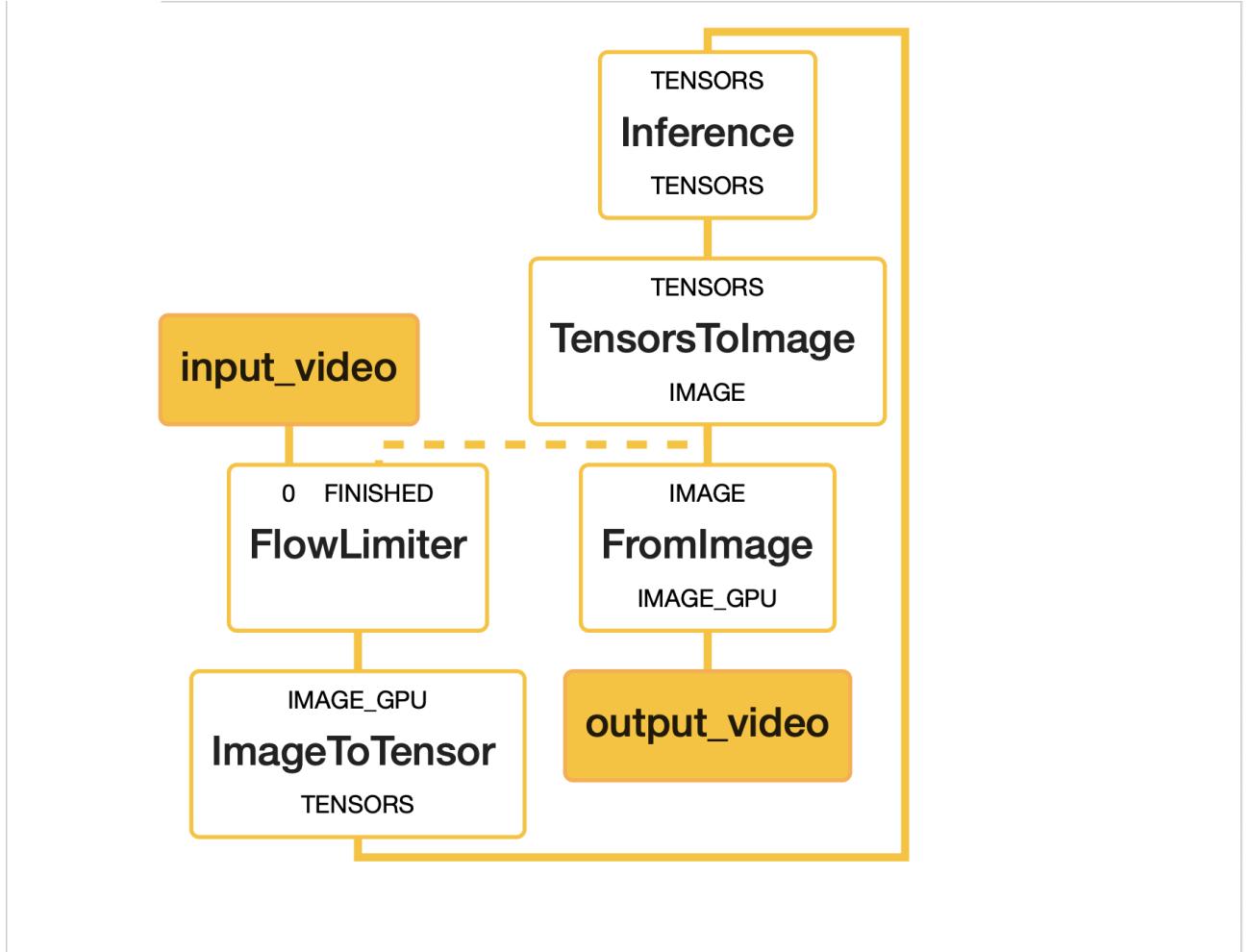


Рисунок 3.1 — Граф конвейера улучшения качества видеопотока

3.2 Конвейер для улучшения качества видеопотока на macOS

Так как для стилизации была выбрана модель whitebox_cartoon_gan и данный инференс будет проходить на CPU и платформе macOS, график будет реализован под landscape разрешение, а именно 360×540 . Таким образом, конвейер будет иметь вид:

```

1 input_stream: "input_video"
2 output_stream: "output_video"
3
4 node {
5     calculator: "FlowLimiterCalculator"
6     input_stream: "input_video"
7     input_stream: "FINISHED:output_image"
8     input_stream_info: {
9         tag_index: "FINISHED"
10        back_edge: true
11    }
12    output_stream: "throttled_input_video"
13}
14
15 node {

```

```

16    calculator: "ImageToTensorCalculator"
17    input_stream: "IMAGE: throttled_input_video"
18    output_stream: "TENSORS:image_tensor"
19    options: {
20        [mediapipe.ImageToTensorCalculatorOptions.ext] {
21            output_tensor_width: 640
22            output_tensor_height: 360
23            keep_aspect_ratio: false
24            output_tensor_float_range {
25                min: -1.0
26                max: 1.0
27            }
28            border_mode: BORDER_ZERO
29        }
30    }
31 }
32
33 node {
34     calculator: "InferenceCalculator"
35     input_stream: "TENSORS:image_tensor"
36     output_stream: "TENSORS:denoised_tensor"
37     options: {
38         [mediapipe.InferenceCalculatorOptions.ext] {
39             model_path: "//mediapipe/models/whitebox_cartoon_gan_360x640_fp16.
40                         tflite"
41             delegate { xnnpack {} }
42         }
43     }
44 }
45 node {
46     calculator: "TensorsToImageCalculator"
47     input_stream: "TENSORS:denoised_tensor"
48     output_stream: "IMAGE:output_image"
49     options: {
50         [mediapipe.TensorsToImageCalculatorOptions.ext] {
51             input_tensor_float_range {
52                 min: -1.0
53                 max: 1.0
54             }
55         }
56     }
57 }
58
59 node {
60     calculator: "FromImageCalculator"
61     input_stream: "IMAGE:output_image"
62     output_stream: "IMAGE_CPU:output_video"
63 }
```

Листинг 3.7: Граф для стилизации на macOS

3.3 Конвейер для улучшения качества видеопотока на iOS и Linux

Так как для стилизации была выбрана модель whitebox_cartoon_gan и данный инференс будет проходить на GPU, на платформах Linux и iOS (граф для них абсолютно одинаковый), граф будет реализован под portrait разрешение, а именно 540×360 . Таким образом, конвейер будет иметь вид:

```
1 input_stream: "input_video"
2 output_stream: "output_video"
3
4 node {
5     calculator: "FlowLimiterCalculator"
6     input_stream: "input_video"
7     input_stream: "FINISHED:output_image"
8     input_stream_info: {
9         tag_index: "FINISHED"
10        back_edge: true
11    }
12    output_stream: "throttled_input_video"
13}
14
15 node {
16     calculator: "ImageToTensorCalculator"
17     input_stream: "IMAGE_GPU:throttled_input_video"
18     output_stream: "TENSORS:image_tensor"
19     options: {
20         [mediapipe.ImageToTensorCalculatorOptions.ext] {
21             output_tensor_width: 360
22             output_tensor_height: 640
23             keep_aspect_ratio: false
24             output_tensor_float_range {
25                 min: -1.0
26                 max: 1.0
27             }
28             border_mode: BORDER_ZERO
29         }
30     }
31 }
32
33 node {
34     calculator: "InferenceCalculator"
35     input_stream: "TENSORS:image_tensor"
36     output_stream: "TENSORS:denoised_tensor"
37     options: {
38         [mediapipe.InferenceCalculatorOptions.ext] {
39             model_path: "//mediapipe/models/whitebox_cartoon_gan_640x360_fp16.
40             tflite"
41             delegate { GPU {} }
42         }
43     }
44 }
45 node {
46     calculator: "TensorsToImageCalculator"
47     input_stream: "TENSORS:denoised_tensor"
48     output_stream: "IMAGE:output_image"
49     options: {
```

```
50 [ mediapipe.TensorsToImageCalculatorOptions.ext ] {
51     input_tensor_float_range {
52         min: -1.0
53         max: 1.0
54     }
55 }
56 }
57 }
58
59 node {
60     calculator: "FromImageCalculator"
61     input_stream: "IMAGE:output_image"
62     output_stream: "IMAGE_GPU:output_video"
63 }
```

Листинг 3.8: Граф для стилизации на Linux и iOS

ГЛАВА 4

РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЙ ДЛЯ ЗАПУСКА ГРАФОВ

4.1 Запуск графа на macOS и Linux

Для запуска графов на macOS и Linux необходимо реализовать простейшее приложение, которое будет принимать кадр с камеры и выводить его на экран. Разница между реализациями на macOS и Linux будет заключаться лишь в том, что на первой платформе в граф будет передано изображение из CpuBuffer'a, а именно mediapipe::ImageFrame, а на второй из mediapipe::GpuBuffer'a.

В самом начале необходимо загрузить и инициализировать конвейер. Он будет передаваться посредством указания пути в параметрах при запуске тестового приложения. Далее подгружается камера. Эта часть общая у обеих платформ. Код приведен ниже.

```
1 std :: string CalculatorGraphConfigContents;
2 MP_RETURN_IF_ERROR( mediapipe :: file :: GetContents(
3     absl :: GetFlag(FLAGS_calculator_graph_config_file) ,
4     &CalculatorGraphConfigContents));
5 ABSL_LOG(INFO) << "Get calculator graph config contents: "
6             << CalculatorGraphConfigContents;
7 mediapipe :: CalculatorGraphConfig Config =
8     mediapipe :: ParseTextProtoOrDie<mediapipe :: CalculatorGraphConfig>(
9         CalculatorGraphConfigContents);
10
11 ABSL_LOG(INFO) << "Initialize the calculator graph." ;
12 mediapipe :: CalculatorGraph Graph;
13 MP_RETURN_IF_ERROR(Graph.Initialize(Config));
14
15 ABSL_LOG(INFO) << "Initialize the camera or load the video." ;
16 cv :: VideoCapture capture;
17 capture.open(0);
18 RET_CHECK(capture.isOpened());
19
20 cv :: namedWindow(kWindowName, /*flags=WINDOW_AUTOSIZE*/ 1);
21 #if (CV_MAJOR_VERSION >= 3) && (CV_MINOR_VERSION >= 2)
22 capture.set(cv :: CAP_PROP_FRAME_WIDTH, 640);
23 capture.set(cv :: CAP_PROP_FRAME_HEIGHT, 480);
24 capture.set(cv :: CAP_PROP_FPS, 30);
25#endif
```

Листинг 4.1: Инициализация графа и камеры

Далее на Linux необходимо инициализировать GPU. Для этого создаются GpuResources и GpuHelper. Код приведен ниже.

```
1 ABSL_LOG(INFO) << "Initialize the GPU." ;
2 MP_ASSIGN_OR_RETURN(auto GpuResources , mediapipe :: GpuResources :: Create());
3 MP_RETURN_IF_ERROR(Graph.SetGpuResources(std :: move(GpuResources)));
```

```

4 mediapipe::GlCalculatorHelper GpuHelper;
5 GpuHelper.InitializeForTest(Graph.GetGpuResources().get());

```

Листинг 4.2: Инициализация GPU на Linux

Затем идет общая часть для двух платформ: запускается граф. Для этого создается Poller. С его помощью данные из выходного потока графа будут возвращены в приложение. В данном случае он следит за потоком с названием "output_video". Затем происходит непосредственный запуск графа. Так как в графе нет input_side_packets, то в функцию StartRun не передается никаких параметров.

```

1 ABSL_LOG(INFO) << "Start running the calculator graph." ;
2 MP_Assign_OR_RETURN(mediapipe::OutputStreamPoller Poller,
3                      Graph.AddOutputStreamPoller("output_video"));
4 MP_RETURN_IF_ERROR(Graph.StartRun({}));

```

Листинг 4.3: Запуск графа

Далее начинается цикл обработки кадров с камеры. Так как модель whitebox_cartoon_gan [2] принимает на вход изображение в формате BGR, и кадр с камеры захватывается с помощью OpenCV, переводить кадр из RGB в BGR не нужно. Из-за того, что выполнение цикла может начаться до окончательной инициализации камеры, пустые кадры необходимо пропустить. Далее кадр поворачивается вертикально на 180 градусов (особенность OpenCV) и передается в функцию SendFrameToGraph. Реализация этой функции будет отличаться на macOS и Linux. Рассмотрим её позже.

После выполнения данной функции, возвращается кадр из графа. Для этого используется GetOutput. Эта функция также будет отличаться на каждой из платформ.

В конце цикла выводится на экран полученный из графа кадр и стоит проверка на закрытие приложения. Код всего цикла приведен ниже.

```

1 ABSL_LOG(INFO) << "Start grabbing and processing frames." ;
2 bool GrabFrames = true;
3 while (GrabFrames)
4 {
5     cv::Mat CameraFrame;
6     capture >> CameraFrame;
7     if (CameraFrame.empty())
8     {
9         ABSL_LOG(INFO) << "Ignore empty frames from camera." ;
10        continue;
11    }
12    cv::flip(CameraFrame, CameraFrame, /*flip code=HORIZONTAL*/ 1);
13
14    SendFrameToGraph(Graph, CameraFrame);
15
16    cv::Mat OutputFrameMat = GetOutput(Poller);
17    cv::imshow("Video Stylisation", OutputFrameMat);
18    const int PressedKey = cv::waitKey(5);
19    if (PressedKey >= 0 && PressedKey != 255) GrabFrames = false;
20}

```

Листинг 4.4: Цикл обработки кадров с камеры

Реализация функции SendFrameToGraph на macOS довольно проста. Кадр, захваченный с камеры, переводится из формата cv::Mat в формат mediapipe::ImageFrame. Затем рассчитывается текущая метка времени (FrameTimestampUs) и вместе с конвертированным кадром передаются в граф.

```

1 void SendFrameToGraph( mediapipe :: CalculatorGraph& Graph , cv :: Mat& CameraFrame )
2 {
3     auto InputFrame = absl :: make_unique<mediapipe :: ImageFrame>(
4         mediapipe :: ImageFormat :: SRGB , CameraFrame . cols , CameraFrame . rows ,
5         mediapipe :: ImageFrame :: kDefaultAlignmentBoundary );
6     cv :: Mat InputFrameMat = mediapipe :: formats :: MatView( InputFrame . get () );
7     CameraFrame . copyTo( InputFrameMat );
8
9     size_t FrameTimestampUs =
10     (double)cv :: getTickCount () / (double)cv :: getTickFrequency () * 1e6 ;
11     MP_RETURN_IF_ERROR( graph . AddPacketToInputStream (
12         "input_video" , mediapipe :: Adopt( InputFrame . release () )
13             . At( mediapipe :: Timestamp( FrameTimestampUs ) ) ) );
14 }
```

Листинг 4.5: Реализация функции SendFrameToGraph на macOS

Реализация SendFrameToGraph на Linux немного отличается. Сперва кадр переводится из cv::Mat в формат mediapipe::ImageFrame. Затем задействованы вспомогательные функции для перевода кадра из формата mediapipe::ImageFrame в mediapipe::GpuBuffer. Далее, с рассчитанной меткой времени, кадр посыпается в граф.

```

1 void SendFrameToGraph( mediapipe :: CalculatorGraph& Graph , cv :: Mat& CameraFrame )
2 {
3     cv :: cvtColor( CameraFrame , CameraFrame , cv :: COLOR_BGR2BGRA );
4
5     auto InputFrame = absl :: make_unique<mediapipe :: ImageFrame>(
6         mediapipe :: ImageFormat :: RGBA , CameraFrame . cols , CameraFrame . rows ,
7         mediapipe :: ImageFrame :: kG1DefaultAlignmentBoundary );
8     cv :: Mat InputFrameMat = mediapipe :: formats :: MatView( InputFrame . get () );
9     CameraFrame . copyTo( InputFrameMat );
10
11     size_t FrameTimestampUs =
12     (double)cv :: getTickCount () / (double)cv :: getTickFrequency () * 1e6 ;
13     MP_RETURN_IF_ERROR(
14         GpuHelper . RunInGlContext (
15             [&InputFrame , &FrameTimestampUs , &Graph , &GpuHelper ]() -> absl :: Status {
16                 auto Texture = GpuHelper . CreateSourceTexture( *InputFrame . get () );
17                 auto GpuFrame = Texture . GetFrame<mediapipe :: GpuBuffer>();
18                 glFlush ();
19                 Texture . Release ();
20                 MP_RETURN_IF_ERROR( Graph . AddPacketToInputStream (
21                     "input_video" , mediapipe :: Adopt( GpuFrame . release () )
22                         . At( mediapipe :: Timestamp( FrameTimestampUs ) ) ) );
23                 return absl :: OkStatus ();
24             }
25         )
26     );
27 }
```

Листинг 4.6: Реализация функции SendFrameToGraph на Linux

Функция GetOutput на macOS забирает пакет из Poller, получает кадр в формате mediapipe::ImageFrame. Переводит из формата mediapipe::ImageFrame в cv::Mat.

```
1 cv :: Mat GetOutput( mediapipe :: OutputStreamPoller& Poller )
2 {
3     mediapipe :: Packet Packet;
4     if (!Poller .Next(&Packet)) break;
5     auto& OutputFrame = Packet .Get<mediapipe :: ImageFrame>();
6
7     return mediapipe :: formats :: MatView(&OutputFrame);
8 }
```

Листинг 4.7: Реализация функции GetOutput на macOS

Функция GetOutput на Linux забирает пакет из Poller, получает кадр в формате mediapipe::GpuBuffer и считывает из framebuffer'a OpenGL контекста в OutputFrame (формат mediapipe::ImageFrame). Затем переводит из формата mediapipe::ImageFrame в cv::Mat. Если выходное изображение имеет 4 канала, переводим из RGBA в RGB.

```
1 cv :: Mat GetOutputGpu( mediapipe :: OutputStreamPoller& Poller )
2 {
3     mediapipe :: Packet Packet;
4     if (!Poller .Next(&Packet)) break;
5     std :: unique_ptr<mediapipe :: ImageFrame> OutputFrame;
6
7     MP_RETURN_IF_ERROR(GpuHelper .RunInGlContext(
8         [&Packet , &OutputFrame , &GpuHelper ]() -> absl :: Status {
9             auto& GpuFrame = Packet .Get<mediapipe :: GpuBuffer>();
10            auto Texture = GpuHelper .CreateSourceTexture(GpuFrame);
11            OutputFrame = absl :: make_unique<mediapipe :: ImageFrame>(
12                mediapipe :: ImageFormatForGpuBufferFormat(GpuFrame .format ()) ,
13                GpuFrame .width () , GpuFrame .height () ,
14                mediapipe :: ImageFrame :: kGlDefaultAlignmentBoundary );
15            GpuHelper .BindFramebuffer(Texture);
16            const auto Info = mediapipe :: GlTextureInfoForGpuBufferFormat(
17                GpuFrame .format () , 0 , GpuHelper .GetGlVersion ());
18            glReadPixels(0 , 0 , texture .width () , texture .height () , Info .gl_
19                format , Info .gl_type , OutputFrame->MutablePixelData ());
20            glFlush ();
21            Texture .Release ();
22            return absl :: OkStatus ();
23        }
24    );
25
26    cv :: Mat OutputFrameMat = mediapipe :: formats :: MatView(OutputFrame .get ());
27    if (OutputFrameMat .channels () == 4)
28        cv :: cvtColor (OutputFrameMat , OutputFrameMat , cv :: COLOR_RGBA2RGB );
29    return OutputFrameMat ;
30 }
```

Листинг 4.8: Реализация функции GetOutput на Linux

После выполнения цикла обработки кадров вызываются функции для закрытия графа, сеанса и камеры.

```
1 ABSL_LOG(INFO) << "Shutting down." ;  
2 MP_RETURN_IF_ERROR(graph.CloseInputStream("input_video")) ;  
3 graph.WaitUntilDone();
```

Листинг 4.9: Закрытие графа и сеанса

Описание кода закончено. Осталось его собрать и протестировать. Ниже приведены примеры работы собранного приложения под MacOS и Linux:



Рисунок 4.1 — Работа приложения на Linux



Рисунок 4.2 — Работа приложения на macOS

4.2 Запуск графа на iOS

Чтобы запустить граф на iOS, разработаем приложение. Для этого необходимо собрать библиотеку на основе MediaPipe. С этой целью был создан следующий файл сборки:

```

1 load("@build_bazel_rules_apple//apple:ios.bzl", "ios_application", "ios_
2   framework")
3 load("@build_bazel_rules_swift//swift:swift.bzl", "swift_library")
4 load(
5   "//mediapipe/examples/ios:bundle_id.bzl",
6   "BUNDLE_ID_PREFIX",
7   "example_provisioning",
8 )
9 licenses(["notice"])  # Apache 2.0
10
11 MIN_IOS_VERSION = "17.6"
12
13 IOS_FAMILIES = [
14   "iphone",
15   "ipad",
16 ]
17
18 ios_framework(
19   name = "Stylising",
20   hdrs = FRAMEWORK_HEADERS,
21   bundle_id = BUNDLE_ID_PREFIX + ".StylisingFramework",
22   bundle_name = "Stylising",
23   families = IOS_FAMILIES,
24   infoplists = [
25     "//mediapipe/examples/ios/common:Info.plist",
26   ],
27   minimum_os_version = MIN_IOS_VERSION,
28   visibility = ["//visibility:public"],
29   deps = [
30     ":ObjcppLib",
31     "@ios_opencv//:OpencvFramework",
32   ],
33 )
34
35 objc_library(
36   name = "ObjcppLib",
37   srcs = [
38     "ObjcppLib.mm",
39   ],
40   hdrs = FRAMEWORK_HEADERS,
41   copts = ["-std=c++17"],
42   data = [
43     "//mediapipe/examples/Stylising/graphs:Stylising_gpu.binarypb",
44     "//mediapipe/models/whitebox_cartoon_gan_540x960_fp16.tflite",
45   ],
46   deps = [
47     "//mediapipe/objc:mediapipe_framework_ios",
48     "//mediapipe/objc:mediapipe_input_sources_ios",
49     "//mediapipe/objc:mediapipe_layer_renderer",
50   ] + select({
51     "//conditions:default": [
52       "//mediapipe/examples/Stylising/graphs:gpu_calculators",
53     ],
54   }),
55 )

```

Листинг 4.10: Файл сборки библиотеки под iOS

Так как на iOS используется Metal, необходимо сделать "прослойку" для

корректной работы графа из фреймворка MediaPipe. Для этого был реализован интерфейс VideoProcessor для связи MediaPipe и iOS приложения. В данном файле присутствует делегат для обработки кадра, интерфейс, инициализирующий, запускающий график, и функция обработки кадра. Также данный интерфейс хранит делегат и метку времени.

```

1 #import <CoreVideo/CoreVideo.h>
2 #import <Foundation/Foundation.h>
3
4 @protocol VideoProcessingDelegate <NSObject>
5 @optional
6 - (void)didProcessFrame:(CVPixelBufferRef)outputBuffer;
7 @end
8
9 @interface VideoProcessor : NSObject
10 - (instancetype)init;
11 - (void)startGraph;
12 - (void)processVideoFrame:(CVPixelBufferRef)imageBuffer;
13 @property(weak, nonatomic) id<VideoProcessingDelegate> delegate;
14 @property(nonatomic) size_t timestamp;
15 @end

```

Листинг 4.11: Интерфейс для связи MediaPipe и iOS приложения

Далее были реализованы данные функции:

1. Функция для очистки всех параметров. Закрывает график, входной и выходной потоки, и удостоверяется, что все ресурсы очищены. Предотвращает утечки памяти.

```

1 #pragma mark - Cleanup methods
2
3 - (void)dealloc {
4     self.mediapipeGraph.delegate = nil;
5     [self.mediapipeGraph cancel];
6     [self.mediapipeGraph closeAllInputStreamsWithError:nil];
7     [self.mediapipeGraph waitUntilDoneWithError:nil];
8 }

```

Листинг 4.12: Функция для очистки всех параметров

2. Функция инициализации графа из ресурсов приложения.

```

1 - (instancetype)init {
2     self = [super init];
3     if (self) {
4         self.mediapipeGraph = [[self class] loadGraphFromResource:kGraphName];
5         self.mediapipeGraph.delegate = self;
6         NSLog(@"%@", kGraphName);
7     }
8     return self;
9 }

```

Листинг 4.13: Функция загрузки графа из ресурсов приложения

3. Вспомогательная функция загрузки графа из ресурсов приложения. После загрузки его инициализируют и запускают. Так как в графике нет input_side_packet, указываются только входной и выходной поток.

```

1 #pragma mark - MediaPipe graph methods
2
3 + (MPPGraph*)loadGraphFromResource:(NSString*)resource {
4     NSError* configLoadError = nil;
5     NSBundle* bundle = [NSBundle bundleForClass:[ self class ]];
6     if (!resource || resource.length == 0) {
7         return nil;
8     }
9     NSURL* graphURL = [ bundle URLForResource:resource withExtension:@"
10      binarypb" ];
11     NSData* data = [ NSData dataWithContentsOfURL:graphURL options:0 error:&
12      configLoadError ];
13     if (!data) {
14         NSLog(@"Failed to load MediaPipe graph config: %@", configLoadError);
15         return nil;
16     }
17     mediapipe::CalculatorGraphConfig config;
18     config.ParseFromArray(data.bytes, data.length);
19     MPPGraph* newGraph = [[MPPGraph alloc] initWithGraphConfig:config];
20
21     [newGraph addFrameOutputStream:kOutputStream outputPacketType:
22      MPPPacketTypePixelBuffer];
23     return newGraph;
24 }
```

Листинг 4.14: Вспомогательная функция загрузки графа из ресурсов приложения

4. Функция запуска графа. Если при запуске графа на стороне MediaPipe возникает ошибка, она выводится в приложении. Иначе выводится лог о старте графа.

```

1 - (void)startGraph {
2     NSError* error;
3     if (![ self.mediapipeGraph startWithError:&error ]) {
4         NSLog(@"Failed to start graph: %@", error);
5     }
6     NSLog(@"Started graph %@", kGraphName);
7 }
```

Листинг 4.15: Функция запуска графа

5. Функция делегата. Оповещает приложение о получении выходного изображения из графа и перенаправляет изображение в приложение.

```

1 #pragma mark - MPPGraphDelegate methods
2
3 - (void)mediapipeGraph:(MPPGraph*)graph
4     didOutputPixelBuffer:( CVPixelBufferRef)pixelBuffer
5         fromStream:( const std::string&)streamName {
6     if (streamName == kOutputStream) {
7         NSLog(@"Received processed frame from output stream");
8         [_delegate didProcessFrame:pixelBuffer];
9     }
10 }
```

Листинг 4.16: Функция делегата

6. Функция обработки кадра. Отправляет кадр во входной поток графа MediaPipe. При возникновении ошибок на любом из этапов, выводит ее в лог и заканчивает работу конвейера.

```
1 #pragma mark - MPPIInputSourceDelegate methods
2
3 - (void)processVideoFrame:(CVPixelBufferRef)imageBuffer {
4     const auto ts =
5         mediapipe::Timestamp(self.timestamp++ * mediapipe::Timestamp::
6             kTimestampUnitsPerSecond);
6     NSError* err = nil;
7     NSLog(@"%@", @"Sending frame %s", ts.DebugString().c_str(),
8             kInputStream);
9     auto sent = [self.mediapipeGraph sendPixelBuffer:imageBuffer
10                 intoStream:kInputStream
11                     packetType:MPPPacketTypePixelBuffer
12                         timestamp:ts
13                         allowOverwrite:NO
14                         error:&err];
15     NSLog(@"%@", "Frame %s", sent ? "sent!" : "not sent.");
16     if (err) {
17         NSLog(@"%@", @"Error sending frame: %@", err);
18     }
19 }
```

Листинг 4.17: Функция обработки кадра

После сборки библиотеки, необходимо применить скрипт patch_framework.sh для получения нужного файла. Для работы с библиотекой, собранной при помощи MediaPipe, надо добавить файл module.modulemap и файлы заголовков. Это необходимо для загрузки всех зависимостей в проект XCode. Ниже представлен код скрипта:

```
1#!/bin/bash
2set -eu
3set -o pipefail
4
5[[ $# -lt 2 ]] && echo "Usage: $0 <path/to/zipped .framework> <hdrs>..." &&
6exit 1
6zipped=$(python -c "import os; print(os.path.realpath('$1'))"); shift
7name=$(basename "$zipped" .zip)
8parent=$(dirname "$zipped")
9named="$parent"/"$name".framework
10
11unzip "$zipped" -d "$parent"
12
13mkdir "$named"/Modules
14cat << EOF >> "$named"/Modules/module.modulemap
15framework module $name {
16    umbrella header "$name.h"
17
18    export *
19    module * { export * }
```

```
20
21 link framework "AVFoundation"
22 link framework "Accelerate"
23 link framework "AssetsLibrary"
24 link framework "CoreFoundation"
25 link framework "CoreGraphics"
26 link framework "CoreImage"
27 link framework "CoreMedia"
28 link framework "CoreVideo"
29 link framework "GLKit"
30 link framework "Metal"
31 link framework "MetalKit"
32 link framework "OpenGL ES"
33 link framework "QuartzCore"
34 link framework "UIKit"
35 }
36 EOF
37
38 cat << EOF >> "$named"/Headers/$name.h
39 #import <Foundation/Foundation.h>
40
41 FOUNDATION_EXPORT double ${name}VersionNumber;
42 FOUNDATION_EXPORT const unsigned char ${name}VersionString[];
43
44 EOF
45 until [[ $# -eq 0 ]]; do
46   printf '#import "'"$1"'"\n' "$1" >> "$named"/Headers/$name.h
47   shift
48 done
```

Листинг 4.18: Скрипт окончательной сборки библиотеки

Далее был создан новый проект в XCode под платформу iOS. В полученном проекте класс AppDelegate был заменен на предоставляемый в MediaPipe. В обновленном классе добавлены функции, которые описывают стандартное поведение приложения, переведенного в фоновый режим. Кроме того, в главном окне приложения устанавливается основной контроллер: StylisingViewController.

```
1 import UIKit
2
3 @UIApplicationMain
4 class AppDelegate: UIResponder, UIApplicationDelegate {
5     lazy var window: UIWindow? = .init(frame: UIScreen.main.bounds)
6
7     func application(
8         application: UIApplication,
9         didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey
10            : Any]?
11     ) -> Bool {
12         window?.rootViewController = StylisingViewController()
13         window?.makeKeyAndVisible()
14         return true
15     }
16
17     func applicationWillResignActive(_ application: UIApplication) {
18     }
19
20     func applicationDidEnterBackground(_ application: UIApplication) {
```

```

20    }
21
22    func applicationWillEnterForeground(_ application: UIApplication) {
23    }
24
25    func applicationDidBecomeActive(_ application: UIApplication) {
26    }
27}

```

Листинг 4.19: Код обновленного класса AppDelegate

Для работы с камерой устройства был реализован класс Camera. В данном классе происходит инициализация камеры. Также реализованы функции для старта сессии работы с камерой (start), для завершения работы камеры (stop). Помимо этого была создана функция для передачи текущего кадра делегату графа MediaPipe и функция смены передней/задней камеры. Так как на вход график ожидает изображение в формате BGR, запрашиваем у камеры именно такой формат.

```

1 import ARKit
2 import AVFoundation
3
4 class Camera: NSObject {
5     enum CameraPosition {
6         case front
7         case back
8     }
9     private(set) var currentPosition: CameraPosition = .front
10
11    lazy var session: AVCaptureSession = .init()
12    private var device: AVCaptureDevice?
13    private var input: AVCaptureDeviceInput?
14    lazy var output: AVCaptureVideoDataOutput = .init()
15
16    override init() {
17        super.init()
18        configureSession(for: currentPosition)
19    }
20
21    private func configureSession(for position: CameraPosition) {
22        session.beginConfiguration()
23        defer { session.commitConfiguration() }
24
25        if let existingInput = input {
26            session.removeInput(existingInput)
27        }
28
29        let newPosition: AVCaptureDevice.Position = position == .front ? .front
30                : .back
31        if let newDevice = AVCaptureDevice.default(.builtInWideAngleCamera, for:
32                        .video, position: newPosition),
33        let newInput = try? AVCaptureDeviceInput(device: newDevice) {
34
35            self.device = newDevice
36            self.input = newInput
37
38            if session.canAddInput(newInput) {
39                session.addInput(newInput)
40            }
41        }
42    }
43
44    func start() {
45        session.startRunning()
46    }
47
48    func stop() {
49        session.stopRunning()
50    }
51
52    func switchCamera() {
53        currentPosition = currentPosition == .front ? .back : .front
54        configureSession(for: currentPosition)
55    }
56
57    func captureImage() {
58        guard let input = input else { return }
59        let image = input.imageBuffer
60        let image = CIImage(cvPixelBuffer: image)
61        let image = image.applyingFilter("CIPhotoEffectSepia")
62        let image = image.jpegData(compressionQuality: 0.5)
63        let image = UIImage(data: image)
64        delegate?.imageCaptured(image)
65    }
66
67    weak var delegate: CameraDelegate?
68}

```

```

39    }
40
41    if session.outputs.isEmpty {
42        output.videoSettings = [kCVPixelBufferPixelFormatTypeKey as String:
43            kCVPixelFormatType_32BGRA]
44        if session.canAddOutput(output) {
45            session.addOutput(output)
46        }
47    }
48
49 func switchCamera() {
50     currentPosition = (currentPosition == .front) ? .back : .front
51     configureSession(for: currentPosition)
52 }
53
54 func setSampleBufferDelegate(_ delegate:
55     AVCaptureVideoDataOutputSampleBufferDelegate) {
56     output.setSampleBufferDelegate(delegate, queue: .main)
57 }
58
59 func start() {
60     if !session.isRunning {
61         session.startRunning()
62     }
63 }
64
65 func stop() {
66     if session.isRunning {
67         session.stopRunning()
68     }
69 }

```

Листинг 4.20: Класс Camera для работы с камерой

Основная логика приложения заключена в классе контроллера StylingViewController. У данного класса присутствуют атрибуты:

1. **camera** — хранит объект камеры и отвечает за передачу кадра из камеры устройства в граф;
2. **displayLayer** — объект типа AVSampleBufferDisplayLayer, который может транслировать кадр из камеры на экран, как только они поступают. Работает напрямую с буфером камеры, что повышает скорость;
3. **videoProcessor** — объект класса VideoProcessor, который отвечает за работу с графом MediaPipe. Для этого был создан класс VideoProcessor;
4. **cameraView** — элемент UI для размещения кадра, полученного напрямую из displayLayer;
5. **imgView** — элемент UI для размещения обработанного VideoProcessor кадра;

6. **switchCameraButton** — кнопка для смены камеры. Доступно два варианта: фронтальная (по умолчанию) и задняя.

```
1 let camera = Camera()
2 let displayLayer: AVSampleBufferDisplayLayer = .init()
3 let videoProcessor: VideoProcessor = VideoProcessor()!
4
5 private lazy var cameraView: UIView = {
6     let view = UIView()
7     view.translatesAutoresizingMaskIntoConstraints = false
8     return view
9 }()
10
11 private lazy var imgView: UIImageView = {
12     let imageView = UIImageView()
13     imageView.translatesAutoresizingMaskIntoConstraints = false
14     imageView.contentMode = .scaleAspectFill
15     return imageView
16 }()
17
18 private lazy var switchCameraButton: UIButton = {
19     let button = UIButton(type: .system)
20     button.translatesAutoresizingMaskIntoConstraints = false
21     button.setTitle("Switch Camera", for: .normal)
22     button.setTitleColor(.white, for: .normal)
23     button.backgroundColor = UIColor.black.withAlphaComponent(0.5)
24     button.layer.cornerRadius = 10
25     button.addTarget(self, action: #selector(didTapSwitchCamera), for: .
26         touchUpInside)
27     return button
28 }()
```

Листинг 4.21: Атрибуты классы StylisingViewController

Далее реализуются основные функции. Первая из них viewDidLoad. Данная функция отвечает за инициализацию и активацию контроллеров и элементов UI. Таким образом настраивается imgView, который занимает весь экран приложения, и кнопка для смены камеры. Также запускаются камера и граф обработки кадра на стороне MediaPipe.

```
1 override func viewDidLoad() {
2     super.viewDidLoad()
3
4     view.addSubview(imgView)
5     view.addSubview(switchCameraButton)
6
7     NSLayoutConstraint.activate([
8         imgView.topAnchor.constraint(equalTo: view.topAnchor),
9         imgView.leadingAnchor.constraint(equalTo: view.leadingAnchor),
10        imgView.trailingAnchor.constraint(equalTo: view.trailingAnchor),
11        imgView.bottomAnchor.constraint(equalTo: view.bottomAnchor),
12
13        switchCameraButton.bottomAnchor.constraint(equalTo: view.
14             safeAreaLayoutGuide.bottomAnchor, constant: -20),
15        switchCameraButton.centerXAnchor.constraint(equalTo: view.centerXAnchor)
16        ,
17        switchCameraButton.widthAnchor.constraint(equalToConstant: 150),
18        switchCameraButton.heightAnchor.constraint(equalToConstant: 50),
19    ])
```

```

17    ])
18
19    camera.setSampleBufferDelegate(self)
20    camera.start()
21    videoProcessor.startGraph()
22    videoProcessor.delegate = self
23 }

```

Листинг 4.22: Функция viewDidLoad

Далее реализованы функции viewDidLoadSubviews и didTapSwitchCamera. Первая удостоверяется, что displayLayer установлен в корректный размер и совпадает с cameraView. didTapSwitchCamera отвечает за нажатие на кнопку смены камеры и, при нажатии, переводит с одной камеры на другую.

```

1 override func viewDidLoadSubviews() {
2     super.viewDidLoad()
3     displayLayer.frame = cameraView.bounds
4 }
5
6 @objc private func didTapSwitchCamera() {
7     camera.switchCamera()
8 }

```

Листинг 4.23: Функции viewDidLoadSubviews и didTapSwitchCamera

Функция captureOutput отвечает за передачу кадров с камеры на конвейер MediaPipe. В данной функции конкретно устанавливается режим работы в portrait формате.

```

1 func captureOutput(
2     _ output: AVCaptureOutput, didOutput sampleBuffer: CMSampleBuffer,
3     from connection: AVCaptureConnection
4 ) {
5     connection.videoOrientation = .portrait
6     displayLayer.enqueue(sampleBuffer)
7     let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer)
8     videoProcessor.processVideoFrame(pixelBuffer)
9 }

```

Листинг 4.24: Функция captureOutput

Функция didProcessFrame забирает кадры из выходного потока графа MediaPipe и выводит их на экран приложения.

```

1 func didProcessFrame(_ outputBuffer: CVPixelBuffer) {
2     DispatchQueue.main.async {
3         let ciImage = CIImage(cvPixelBuffer: outputBuffer)
4         let uiImage = UIImage(ciImage: ciImage)
5         self.imgView.contentMode = .scaleAspectFill
6         self.imgView.image = uiImage
7     }
8 }

```

Листинг 4.25: Функция didProcessFrame

Это был последний класс, который был необходим для работы приложения. Осталось его собрать и протестировать. Ниже приведена работа собранного приложения под iOS:



Рисунок 4.3 — Работа приложения под iOS

Заключение

В ходе работы:

1. Была рассмотрена задача улучшения видеопотока с помощью фреймворка MediaPipe, а также ее применение на прикладном уровне;
2. Сделан краткий обзор фреймворка MediaPipe;
3. Выполнен обзор основных элементов конвейера фреймворка MediaPipe;
4. Разработана общая архитектура конвейера;
5. Реализованы конвейеры для платформ macOS, Linux и iOS;
6. Разработаны приложения для тестирования конвейеров на macOS, Linux и iOS на языках программирования C++ и Swift;
7. Успешно протестированы результаты работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Lugaressi.C., Tang.J., Nash.H., McClanahan.C., Ubweja.E., Hays.M., Zhang.F., Chang.C.-L., Yong.M.G., Lee.J., Chang.W.-T., Hua.W., Georg.M., Grundmann.M. (2019). MediaPipe: A Framework for Perceiving and Processing Reality [Электронный ресурс]: CVPR CV4ARVR Workshop 2019/Google Research. – Mountain View, CA: Google Research, 2019. – Режим доступа: https://static1.squarespace.com/static/5c3f69e1cc8fedbc039ea739/t/5e130ff310a69061a71cbd7c/1578307584840/NewTitle_May1_MediaPipe_CVPR_CV4ARVR_Workshop_2019.pdf. – Дата доступа: 18.12.2024.
2. Wang.X., Yu.J. (2020). Learning to Cartoonize Using White-box Cartoon Representations [Электронный ресурс]: CVPR 2020/ByteDance, The University of Tokyo, Style2Paints Research. – Mountain View, CA: ByteDance, 2020. – Режим доступа: https://openaccess.thecvf.com/content_CVPR_2020/papers/Wang_Learning_to_Cartoonize_Using_White-Box_Cartoon_Representations_CVPR_2020_paper.pdf. – Дата доступа: 18.12.2024.
3. Sharma.S., Kama.S., Bernauer.J., Moroney.L. (2018). Speed up TensorFlow Inference on GPUs with TensorRT [Электронный ресурс]: The TensorFlow Blog / NVidia. – Город издания: NVidia, 2018. – Режим доступа: <https://blog.tensorflow.org/2018/04/speed-up-tensorflow-inference-on-gpus-tensorRT.html>. – Дата доступа: 18.12.2024.
4. Google. (2023). MediaPipe Framework [Electronic resource]: Google for Developers. Режим доступа: <https://developers.google.com/mediapipe/framework>. – Дата доступа: 18.12.2024
5. TensorFlow. (2023). Segmentation [Electronic resource]: TensorFlow Lite. Режим доступа: <https://www.tensorflow.org/lite/examples/segmentation/overview>. – Дата доступа: 18.12.2024
6. bazel.build. Bazel [Electronic resource]. Режим доступа: <https://bazel.build>. – Дата доступа: 18.12.2024