

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

**Обработка видеопотоков с заменой фона с использованием
фреймворка MediaPipe**

Курсовой проект

Бинцаровского Леонида Петровича
студента 3 курса
специальность «информатика»

Научный руководитель:
старший преподаватель
Д. И. Пиршук

Минск, 2023

Реферат

Курсовой проект, 27 стр., 11 иллюстр., 4 источника.

Ключевые слова: сегментация; Visual Studio; C++; MediaPipe; граф; калькулятор; конвейер.

Объекты исследования — задача сегментация кадра видеопотока для изменением фона.

Цель исследования — реализация собственного конвейера и классов для использования фреймворка MediaPipe в среде разработки Visual Studio.

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

В результате исследования был реализован конвейер для обработки видеопотока с заменой изображения, а также классы для работы с фреймворком MediaPipe в среде разработки Visual Studio.

Области применения — процесс обработки видеопотока для изменения фона.

Оглавление

Введение	3
1 Постановка задачи обработки видеопотоков с изменением фона	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 обработки видеопотока для изменения фона	8
3.1 Разработка архитектуры конвейера	8
3.2 Реализация калькуляторов для полученного конвейера	13
4 Реализация классов для работы в среде Visual Studio	17
4.1 Реализация классов	17
4.2 Создание проекта в среде Visual Studio и его настройка	23
Заключение	26
Список использованных источников	27

Введение

В настоящее время обработка видеопотоков и редактирование контента стали неотъемлемой частью современной технологической среды. Одним из направлений в этой области является обработка видеопотоков с заменой фона. Эта технология находит применение в различных областях, начиная с создания увлекательных видеороликов и заканчивая трансляциями виртуальной реальности.

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.

ГЛАВА 1

ПОСТАНОВКА ЗАДАЧИ ОБРАБОТКИ ВИДЕОПОТОКОВ С ИЗМЕНЕНИЕМ ФОНА

Задача обработки видеопотоков для изменения фона будет сводится к другой, более обширной задаче: сегментации изображения (англ. *image segmentation*). Так как сегментация изображения — это одна из основных задач компьютерного зрения (англ. *computer vision*), существует множество подходов её реализации. Выделим основные из них:

1. Выделение краёв;
2. Методы разреза графа;
3. Сегментация с помощью предобученной модели.

На данный момент самым распространенным и точным является подход сегментации фона с помощью предобученной модели. Модели делятся на:

1. **Semantic segmentation.** Разделение на два класса: фон и объект;
2. **Hair segmentation.** Разделение на два класса: фон и волосы;
3. **Multi-class segmentation.** Разделение на шесть классов: фон, волосы, тело, лицо, одежда и аксессуары;
4. **Instance segmentation.** Разделение на $n + 1$ класс: фон и n объектов.

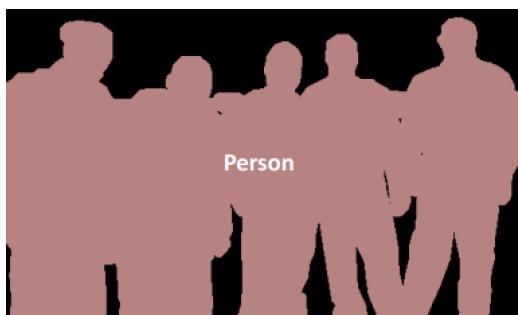


Рисунок 1.1 — Semantic segmentation

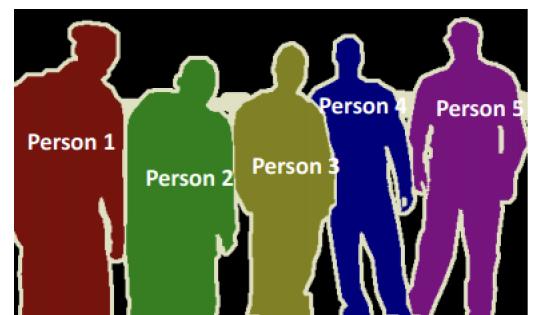


Рисунок 1.2 — Instance segmentation

Для задачи изменения фона на видеопотоке достаточно определить объект (человека) и фон, поэтому воспользуемся моделью semantic segmentation. Для реализации конвейера semantic segmentation воспользуемся фреймворком MediaPipe. А также реализуем C++ классы для работы с фреймворком MediaPipe в среде разработки Visual Studio.

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 руки, 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++ для платформы Windows.

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 определенного 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 внутренний, входной (input) и исходящий (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.

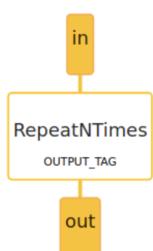


Рисунок 2.1 —
Простейший граф
(визуализация)

```
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.

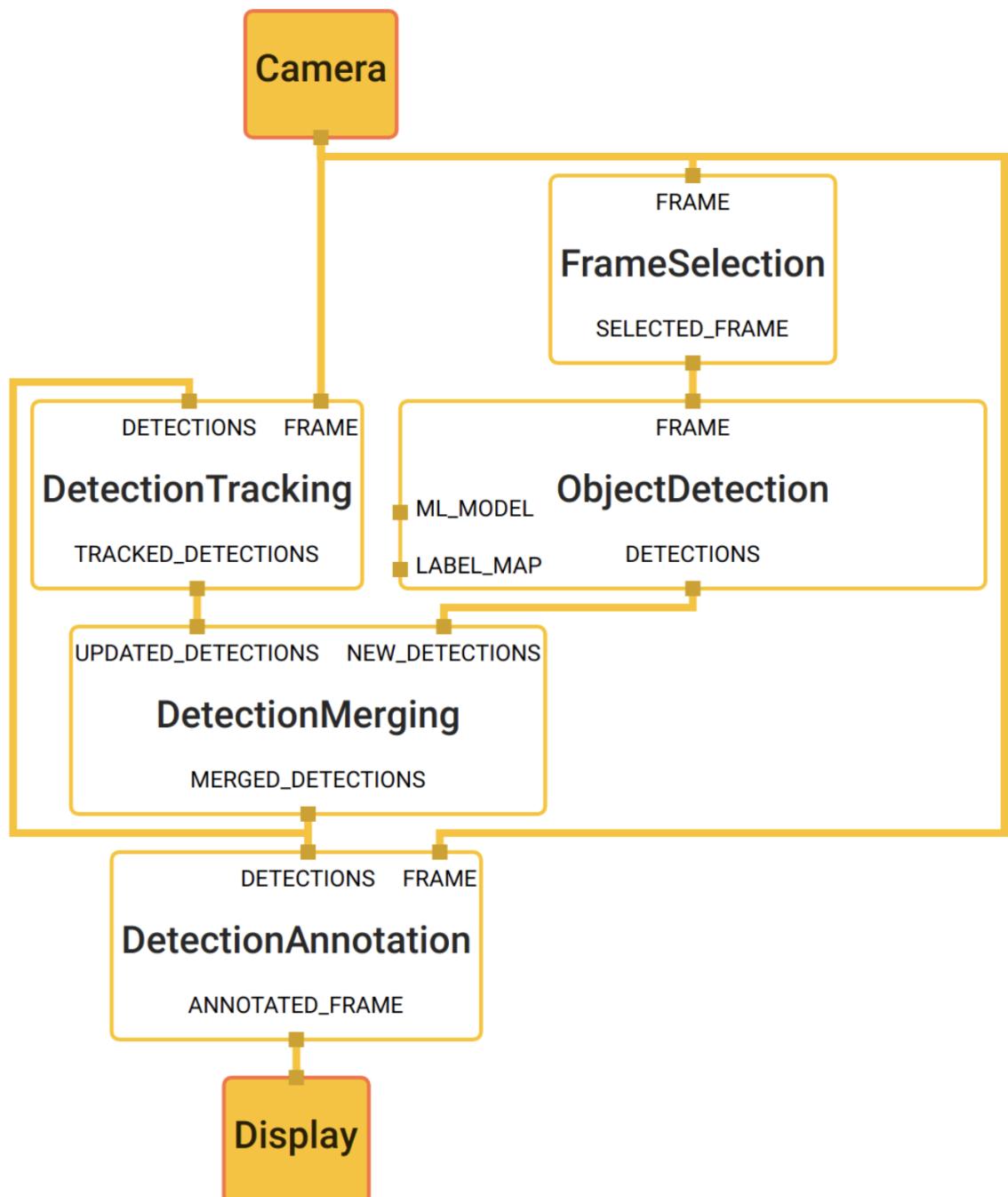


Рисунок 2.3 – Пример конвейера

ГЛАВА 3

РАЗРАБОТКА КОНВЕЙЕРА ОБРАБОТКИ ВИДЕОПОТОКА ДЛЯ ИЗМЕНЕНИЯ ФОНА

3.1 Разработка архитектуры конвейера

Для запуска модели необходимо создать конвейер, который в случае с фраймворком MediaPipe представляет собой граф из некоторого количества калькуляторов. В этом пункте определим архитектуру всего конвейера, а в следующем опишу реализацию некоторых калькуляторов.

На вход в конвейер будет поступать текущее изображение видеопотока и изображение для замены фона. На выходе конвейер будет возвращать текущий кадр с измененным фоном. Фон может быть размыт или заменен на изображение, переданное как параметр. Для semantic segmentation в фраймворке MediaPipe предоставлены две модели:

- **SelfieSegmenter (square)**. Данная модель ожидает на вход изображение размером 256×256 и в большей мере предназначена для приложений в вертикальном расположении экрана;
- **SelfieSegmenter (landscape)**. Данная модель ожидает на вход изображение размером 144×256 и в большей мере предназначена для приложений в горизонтальном расположении экрана.

Для предоставления возможности переключения между моделями, в зависимости от поставленной задачи, дополнительно на входе конвейер будет принимать тип модели в дополнительный пакет (англ. side packet): 0 — SelfieSegmenter (square), 1 — SelfieSegmenter (landscape). Также в еще один дополнительный пакет будет задаваться тип работы конвейера: 0 — замена фона на изображение, 1 — размытие фона.

```
1 # CPU buffer containing the current frame. (type: ImageFrame)
2 input_stream: "input_video"
3
4 # CPU buffer containing the background image. (type: ImageFrame)
5 input_stream: "input_background"
6
7 # Output image with rendered results. (type: ImageFrame)
8 output_stream: "output_video"
9
10 # Type of effect. (type: int)
11 input_side_packet: "TYPE_OF_EFFECT:type_of_effect"
12
13 # Type of model. (type: int)
14 input_side_packet: "MODEL_SELECTION:model_selection"
```

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 переведен 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 последнего калькулятора конвейера. В input_stream_info указывается параметр, которому должен быть равен поток FINISHED, для того чтобы калькулятор пропустил следующее изображение. Если поток FINISHED будет не пустой, калькулятор передаст следующее изображение, иначе они будут отбрасываться.

```

1 node {
2   calculator: "FlowLimiterCalculator"
3   input_stream: "input_video"
4   input_stream: "FINISHED:output_video"
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 изображения маски: 0 — человек, 1 — фон. Для этого применим подграф (вспомогательный граф, который используется как калькулятор) SelfieSegmentationSubgraph.

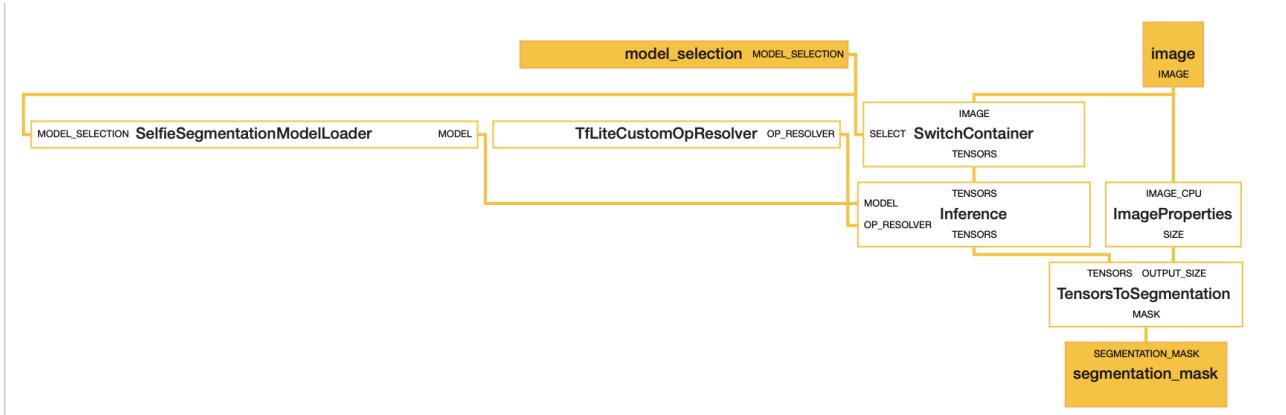


Рисунок 3.1 — Подграф для сегментации изображения

На рис. 3.1 представлена архитектура данного подграфа. В граф SelfieSegmentationSubgraph на вход передаются текущий кадр (в текущей ре-

ализации это throttled_input_video и тип модели). На выходе подграф возвращает изображение маски сегментации. Для обработки используются калькуляторы:

- SwitchContainer, который обрабатывает тип модели и, в зависимости от полученных данных (1 или 0), преобразует полученное изображение, при помощи калькулятора ImageToTensorCalculator, в 3-мерный тензор размера $256 \times 256 \times C$ или $144 \times 256 \times C$, где C — количество каналов в изображении;
- TfLiteCustomOpResolverCalculator, используется для генерации единственного дополнительного пакета (side packet) с именем OP_RESOLVER, который содержит TensorFlow Lite операционный разрешитель (op resolver), поддерживающий пользовательские операции (custom ops), необходимые для модели, используемой в данном графе;

```

1 node {
2   calculator: "TfLiteCustomOpResolverCalculator"
3   output_side_packet: "OP_RESOLVER:op_resolver"
4 }
```

Листинг 3.3: Вызов калькулятора TfLiteCustomOpResolverCalculator в подграфе

- SelfieSegmentationModelLoader — загружает предобученную модель в дополнительный пакет в зависимости от заданного типа (1 или 0);

```

1 node {
2   calculator: "SelfieSegmentationModelLoader"
3   input_side_packet: "MODEL_SELECTION:model_selection"
4   output_side_packet: "MODEL:model"
5 }
```

Листинг 3.4: Вызов калькулятора SelfieSegmentationModelLoader в подграфе

- InferenceCalculator — калькулятор, отвечающий за вывод (англ. inference) модели. На вход ему предоставляется тензор, созданный в калькуляторе SwitchContainer; дополнительные пакеты, содержащие предобученную модель и операционный разрешитель. Также в данном калькуляторе указывается имя библиотеки (XNNPack) для оптимизированного выполнения на CPU. На выход данный калькулятор возвращает тензор $256 \times 256 \times 2$ или же $144 \times 256 \times 2$, где в последнем измерении первый канал представляет собой сегментацию фона, а второй канал — сегментацию переднего плана;

```

1 node {
2   calculator: "InferenceCalculator"
3   input_stream: "TENSORS:input_tensors"
4   output_stream: "TENSORS:output_tensors"
```

```

5   input_side_packet: "MODEL: model"
6   input_side_packet: "OP_RESOLVER: op_resolver"
7   options: {
8     [ mediapipe.InferenceCalculatorOptions.ext] {
9       delegate {
10         xnnpack {}
11     }
12   }
13 }
14 }
```

Листинг 3.5: Вызов калькулятора InferenceCalculator в подграфе

- ImagePropertiesCalculator получает на вход стартовое изображение и возвращает его размер;

```

1 node {
2   calculator: "ImagePropertiesCalculator"
3   input_stream: "IMAGE_CPU:image"
4   output_stream: "SIZE:input_size"
5 }
```

Листинг 3.6: Вызов калькулятора ImagePropertiesCalculator в подграфе

- TensorsToSegmentationCalculator получает на вход обработанный тензор из калькулятора SwitchContainer, маску из калькулятора InferenceCalculator, размер входного изображения из ImagePropertiesCalculator, объединяет все вместе и возвращает изображение маски для текущего кадра.

```

1 node {
2   calculator: "TensorsToSegmentationCalculator"
3   input_stream: "TENSORS:output_tensors"
4   input_stream: "OUTPUT_SIZE:input_size"
5   output_stream: "MASK:segmentation_mask"
6   options: {
7     [ mediapipe.TensorsToSegmentationCalculatorOptions.ext] {
8       activation: NONE
9     }
10   }
11 }
```

Листинг 3.7: Вызов калькулятора TensorsToSegmentationCalculator в графе конвейера

Далее происходит обработка полученных изображений. Сначала вызывается калькулятор SegmentationSmoothingCalculator, который предназначен для уменьшения дрожания края маски. Для этого в данный калькулятор передается маска сегментации, полученная из подграфа SelfieSegmentationSubgraph, маска сегментации, сохраненная из предыдущей операции этого же калькулятора и, в качестве константного параметра, коэффициент, с которым будет применяться процесс сглаживания двух масок. На выход данный калькулятор возвращает объединенную маску сегментации с более четкими краями.

```

1 node {
2   calculator: "SegmentationSmoothingCalculator"
3   input_stream: "MASK:segmentation_mask"
4   input_stream: "MASK_PREVIOUS:prev_filtered_segmentation_mask"
5   output_stream: "MASK_SMOOTHED:filtered_segmentation_mask"
6   node_options {
7     [ type.googleapis.com/mediapipe.SegmentationSmoothingCalculatorOptions ] {
8       combine_with_previous_ratio: 0.7
9     }
10   }
11 }
```

Листинг 3.8: Вызов калькулятора SegmentationSmoothingCalculator в графе конвейера

Для сохранения текущей маски, которая будет передана на следующую операцию калькулятора SegmentationSmoothingCalculator, вызывается калькулятор PreviousLoopbackCalculator.

```

1 node {
2   calculator: "PreviousLoopbackCalculator"
3   input_stream: "MAIN:segmentation_mask" # Tick signal
4   input_stream: "LOOP:filtered_segmentation_mask"
5   input_stream_info: {
6     tag_index: "LOOP"
7     back_edge: true
8   }
9   output_stream: "PREV_LOOP:prev_filtered_segmentation_mask"
10 }
```

Листинг 3.9: Вызов калькулятора PreviousLoopbackCalculator в графе конвейера

Чтобы получить маску изображения в формате, подходящем для дальнейшей обработки, используется калькулятор FromImageCalculator. На вход данному калькулятору передается маска сегментации из калькулятора SegmentationSmoothingCalculator, а на выход данный калькулятор отдает маску сегментации, преобразованную в тип ::mediapipe::ImageFrame.

```

1 node: {
2   calculator: "FromImageCalculator"
3   input_stream: "IMAGE:filtered_segmentation_mask"
4   output_stream: "IMAGE_CPU:temp_mask"
5 }
```

Листинг 3.10: Вызов калькулятора FromImageCalculator в графе конвейера

На данный момент все готово для вызова калькулятора, который непосредственно и будет изменять фон. Для этого вызовем калькулятор SwitchContainer, который, в зависимости от выбранного режима работы, будет запускать калькулятор BackgroundMaskingCalculator или BackgroundBlurringCalculator. Также в данный калькулятор передается маска сегментации из FromImageCalculator, текущий кадр из FlowLimiterCalculator и изображение фона. На выход данный калькулятор возвращает изображение с измененным фоном (замененным на изображение или размытым).

```

1 node {
2   calculator: "SwitchContainer"
3   input_side_packet: "SELECT:type_of_effect"
4   input_stream: "IMAGE CPU: throttled_input_video"
5   input_stream: "MASK CPU: temp_mask"
6   input_stream: "BACKGROUND CPU: input_background"
7   output_stream: "OUTPUT_VIDEO: output_video"
8   options: {
9     [ mediapipe.SwitchContainerOptions.ext ] {
10       select: 0
11       contained_node: {
12         calculator: "BackgroundMaskingCalculator"
13       }
14       contained_node: {
15         calculator: "BackgroundBlurringCalculator"
16       }
17     }
18   }
19 }

```

Листинг 3.11: Вызов калькулятора SwitchContainer в графе конвейера

На данном калькуляторе описание архитектуры конвейера заканчивается. Для лучшего понимания полноценной архитектуры прилагается визуализированный граф конвейера (рис 3.2)

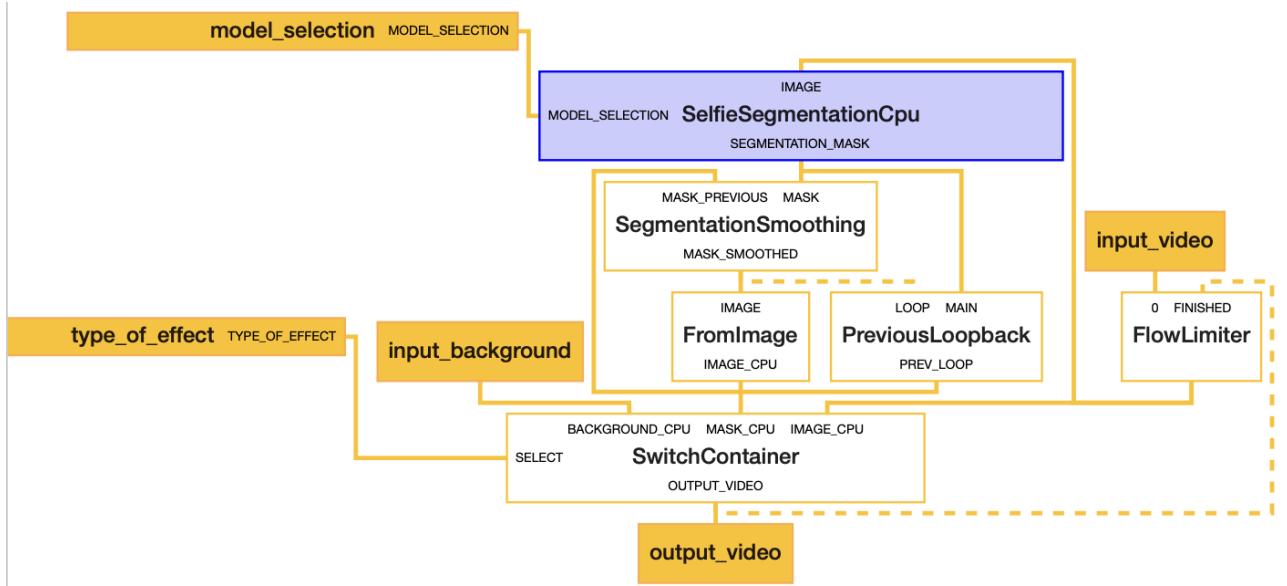


Рисунок 3.2 — Граф конвейера обработки видеопотока

3.2 Реализация калькуляторов для полученного конвейера

После формирования архитектуры конвейера необходимо создать калькуляторы. Некоторые из основных калькуляторов были реализованы и представлены вместе с фреймворком MediaPipe.

Начнем с калькулятора BackgroundMaskingCalculator. Данный калькулятор представляет собой класс C++, реализующий интерфейс CalculatorBase. В данном классе присутствуют следующие функции:

- **::mediapipe::Status GetContract(CalculatorContract* cc).**

```

1 static ::mediapipe::Status GetContract(CalculatorContract* cc) {
2     RET_CHECK(cc->Inputs().HasTag("IMAGE_CPU"));
3     RET_CHECK(cc->Inputs().HasTag("MASK_CPU"));
4     RET_CHECK(cc->Outputs().HasTag("OUTPUT_VIDEO"));
5     RET_CHECK(cc->Inputs().HasTag("BACKGROUND_CPU"));
6     cc->Inputs().Tag("IMAGE_CPU").Set<ImageFrame>();
7     cc->Inputs().Tag("MASK_CPU").Set<ImageFrame>();
8     cc->Outputs().Tag("OUTPUT_VIDEO").Set<ImageFrame>();
9     cc->Inputs().Tag("BACKGROUND_CPU").Set<ImageFrame>();
10
11     return ::mediapipe::OkStatus();
12 }
```

Листинг 3.12: Функция GetContract для калькулятора BackgroundMaskingCalculator

Данная функция описывает входные и выходные параметры калькулятора. Для этого используется макрос RET_CHECK(), который проверяет задан ли указанный входной или выходной поток. Если данный поток не задан, будет выведено соответствующее сообщение. В строках 6-9 устанавливается тип данных, который ожидается на данном потоке. Если при запуске произойдет несоответствие типам данных, также будет выведена ошибка;

- **::mediapipe::Status Process(CalculatorContext* cc).**

```

1 ::mediapipe::Status Process(CalculatorContext* cc) final {
2     const auto& input_img =
3         cc->Inputs().Tag("IMAGE_CPU").Get<ImageFrame>();
4     cv::Mat input_mat = formats::MatView(&input_img);
5
6     const auto& background_img =
7         cc->Inputs().Tag("BACKGROUND_CPU").Get<ImageFrame>();
8     cv::Mat background = formats::MatView(&background_img);
9
10    const auto& mask_img =
11        cc->Inputs().Tag("MASK_CPU").Get<ImageFrame>();
12    cv::Mat mask_mat = formats::MatView(&mask_img);
13
14    cv::cvtColor(background, background, cv::COLOR_RGB2BGR);
15    cv::resize(background, background, input_mat.size());
16
17    mask_mat.convertTo(mask_mat, CV_8UC3);
18    cv::blur(mask_mat, mask_mat, cv::Size(15, 15));
19    cv::threshold(mask_mat, mask_mat, 0.5, 255, cv::THRESH_BINARY);
20
21    background.copyTo(input_mat, 1.0 - mask_mat);
22    cv::Mat output_image = input_mat;
23
24    auto output_frame = absl::make_unique<ImageFrame>(
```

```

25     input_img.Format() , input_img.Width() , input_img.Height() );
26
27     cv::Mat output_mat =
28         mediapipe::formats::MatView(output_frame.get());
29     output_image.copyTo(output_mat);
30
31     cc->Outputs()
32         .Tag("OUTPUT_VIDEO")
33         .Add(output_frame.release() , cc->InputTimestamp());
34
35     return ::mediapipe::OkStatus();
36 }
```

Листинг 3.13: Функция Process для калькулятора BackgroundMaskingCalculator

Эта функция обрабатывает поступившие пакеты. В данном случае это замена фона на изображение. Изначально при помощи функций в строках 2-4 извлекается ссылка на входное изображение (input_img) в формате ImageFrame и конвертируется в тип данных cv::Mat (input_mat). Тот же подход используется и для изображения фона (background_mat), и для маски сегментации (mask_mat). Далее идет обработка полученных данных: изображение фона переводится в формат BGR, применяемому в библиотеке OpenCV, и приводится к размеру входного кадра видеопотока. После этого применяется размытие и бинаризация маски сегментации (строки 18-19). Затем полученная маска применяется к текущему кадру и фону, вследствие чего получаем изображение с замененным фоном, которое переводим в формат ImageFrame и передаем на выходной поток (строки 27-33).

Весь класс располагается в .cpp файле, заголовочный файл не требуется, регистрацию класса производит макрос REGISTER_CALCULATOR.

Помимо самого кода калькулятора необходимо определить proto-файл с конфигурацией калькулятора. Для BackgroundMaskingCalculator это стандартный путь к изображению фона.

```

1 syntax = "proto2";
2 package mediapipe;
3 import "mediapipe/framework/calculator.proto";
4 message BackgroundMaskingCalculatorOptions {
5     optional string path_to_background = 1 [default = "background.jpg"];
6 }
```

Листинг 3.14: Proto-файл с конфигурацией калькулятора

Для калькулятора BackgroundBlurringCalculator весь код аналогичен, за исключением обработки фона: изначально применяется маска сегментации к исходному кадру, вследствие чего получается сегментированное изображение фона. Затем к данному изображению применяется размытие. Далее код совпадает с кодом калькулятора BackgroundBlurringCalculator.

```
1 input_mat.copyTo(background, 1.0 - mask_mat);
2 cv::GaussianBlur(mask_mat, mask_mat, cv::Size(3, 3), 7);
3 cv::GaussianBlur(background, background, cv::Size(19, 19), 7);
4
5 background.copyTo(input_mat, 1.0 - mask_mat);
```

Листинг 3.15: Код измененной части относительно калькулятора BackgroundBlurringCalculator

ГЛАВА 4

РЕАЛИЗАЦИЯ КЛАССОВ ДЛЯ РАБОТЫ В СРЕДЕ VISUAL STUDIO

4.1 Реализация классов

Для сборки проекта на базе фреймворка MediaPipe необходимо использовать систему сборки `bazel`. В связи с этим использовать фреймфорк MediaPipe в среде разработки Visual Studio и, тем более, собрать проект невозможно. Поэтому была поставлена задача создать классы для запуска фреймворка MediaPipe в Visual Studio.

Первым делом был реализован интерфейс `gmod_api.h`:

```
1 #pragma once
2 #include<string>
3 #include<functional>
4
5 #ifdef _WIN32
6 #define DLLEXPORT __declspec(dllexport)
7 #else
8 #define DLLEXPORT
9 #endif
10
11 DLLEXPORT class IGMOD {
12 public:
13     virtual bool getOverlay() = 0;
14     virtual void setOverlay(bool x) = 0;
15
16     virtual void setCameraProps(const int& camId,
17                                 const int& camResX,
18                                 const int& camResY,
19                                 const int& camFps) = 0;
20
21     virtual void start(cv::Mat& camera_frame_raw, bool& _load_flag) = 0;
22     virtual void init(const std::string& filename,
23                       cv::VideoCapture& capture,
24                       const std::string& path = "background.jpg",
25                       const int& typeOfEffect = 0,
26                       const int& typeOfModel = 1) = 0;
27     virtual void stop() = 0;
28 };
29
30 DLLEXPORT IGMOD* CreateGMOD();
```

Листинг 4.1: Интерфейс `gmod_api`

Далее был создан заголовок класса `gmod_core.h`, который реализует интерфейс `gmod_api.h`:

```
1 class GMOD : public IGMOD {
2 public:
3     virtual bool getOverlay() override;
4     virtual void setOverlay(bool x) override;
```

```

5  virtual void setCameraProps( const int& camId ,
6                                const int& camResX ,
7                                const int& camResY ,
8                                const int& camFps) override ;
9
10 virtual void start(cv::Mat& camera_frame_raw , bool& _load_flag) override ;
11 virtual void init( const std::string& filename ,
12                     cv::VideoCapture& capture ,
13                     const std::string& path = "background.jpg" ,
14                     const int& typeOfEffect = 0 ,
15                     const int& typeOfModel = 1) override ;
16
17
18 private:
19     absl::Status _runMPPGraph(cv::Mat& camera_frame_raw , bool& _load_flag) ;
20     absl::Status _initMPPGraph(cv::VideoCapture& capture) ;
21
22 private:
23     int camId ;
24     int camFps ;
25     int camResX ;
26     int camResY ;
27     int typeOfModel ;
28     int typeOfEffect ;
29     bool showOverlay ;
30     cv::Mat background ;
31     std::string graphFilename ;
32     std::string backgroundFilename ;
33
34     std::unique_ptr<mediapipe::OutputStreamPoller> output_poller ;
35
36     const char kInputStream[12] = "input_video" ;
37     const char kOutputStream[13] = "output_video" ;
38     const char kInputBackground[17] = "input_background" ;
39
40 public:
41     std::shared_ptr<mediapipe::CalculatorGraph> _graph ;
42 };

```

Листинг 4.2: Класс gmod_core

У класса есть как private атрибуты, так и один public атрибут. В атрибутах присутствуют следующие переменные:

- **camId** — переменная, отвечающая за индекс камеры. Тип данной переменной — int;
- **camFps** — переменная, отвечающая за количество кадров в секунду (англ. frames per second, fps), получаемых из камеры. Тип данной переменной — int;
- **camResX** и **camResY**, отвечающие за требуемый размер получаемого кадра из камеры. Тип данных переменных — int;
- **typeOfModel** — переменная, отвечающая за тип модели, которую необходимо использовать. Тип данной переменной — int. Значения должны быть 0 или 1;

- **typeOfEffect** — переменная, отвечающая за режим работы конвейера. Тип данной переменной — int. Значения должны быть 0 — замена фона или 1 — размытие фона;
- **showOverlay** — переменная, разрешающая использовать преобразования над фоном. Тип данной переменной — bool;
- **background** — переменная, хранящая изображение виртуального фона. Тип данной переменной — cv::Mat;
- **graphFilename** — переменная, хранящая путь до графа конвейера. Тип данной переменной — std::string;
- **backgroundFilename** — переменная, хранящая путь до изображения фона. Тип данной переменной — std::string;
- **output_poller** — переменная, хранящая указатель на информацию из выходного потока конвейера. Тип данной переменной — std::unique_ptr<mediapipe::OutputStreamPoller>;
- **kInputStream** — константная переменная, хранящая ожидаемое имя входного потока в конвейер, содержащего текущий кадр. Тип данной переменной — массив char;
- **kOutputStream** — константная переменная, хранящая ожидаемое имя выходного потока из конвейера. Тип данной переменной — массив char;
- **kInputBackground** — константная переменная, хранящая ожидаемое имя входного потока в конвейер, содержащего изображение фона. Тип данной переменной — массив char;
- **_graph** — переменная, хранящая указатель на переменную графа конвейера. Тип данной переменной — std::shared_ptr<mediapipe::CalculatorGraph>;

Так как в большинстве своем в фреймворке MediaPipe используются макросы, которые возвращают ошибки типа absl::Status из фреймворка MediaPipe и их нельзя обработать напрямую в коде программы в Visual Studio, были реализованы вспомогательные функции в private, выполняющие основной функционал, и основные функции (будут вызываться в основной программе), которые являются обертками функций из private и обрабатывают возможные ошибки. Сначала рассмотрим основные функции (public), а затем реализацию вспомогательных (private).

Функция **bool getOverlay()** возвращает true, если установлен флаг для обработки изображения. False в противном случае;

```

1 bool GMOD::getOverlay() {
2     return showOverlay;
3 }
```

Листинг 4.3: Функция getOverlay()

Функция **void setOverlay(bool x)** устанавливает значение флага для обработки изображения в режим *x*;

```

1 void GMOD::setOverlay(bool x) {
2     this->showOverlay = x;
3 }
```

Листинг 4.4: Функция setOverlay()

void setCameraProps(const int& camId, const int& camResX, const int& camResY, const int& camFps) устанавливает четыре параметра камеры: индекс, расширение по $X \times Y$ и количество кадров в секунду (англ. fps);

```

1 void GMOD::setCameraProps(const int& camId,
2                             const int& camResX,
3                             const int& camResY,
4                             const int& camFps) {
5     this->camId = camId;
6     this->camResX = camResX;
7     this->camResY = camResY;
8     this->camFps = camFps;
9 }
```

Листинг 4.5: Функция setCameraProps()

Функция **void init(...)**, инициализирующая основные параметры графа и камеры, вызывая вспомогательную функцию и обрабатывая возможные предупреждения и ошибки. На вход принимает путь до файла графа, переменную видеопотока и, при необходимости, путь до изображения фона, режим работы конвейера, тип модели.

```

1 void GMOD::init(const std::string& filename ,
2                  cv::VideoCapture& capture ,
3                  const std::string& path ,
4                  const int& typeOfEffect ,
5                  const int& typeOfModel) {
6     this->graphFilename = filename;
7     this->backgroundFilename = path;
8     this->typeOfEffect = (typeOfEffect != 0 && typeOfEffect != 1 ? 1 :
9                             typeOfEffect);
10    this->typeOfModel = (typeOfModel != 0 && typeOfModel != 1 ? 1 :
11                           typeOfModel);
12    auto status = this->initMPPGraph(capture);
13    if (!status.ok()) {
14        std::string msg(status.message());
15        LOG(ERROR) << (msg);
16    }
17 }
```

Листинг 4.6: Функция init()

Функция `void start(cv::Mat& camera_frame_raw, bool& _load_flag)` запускает конвейер графа. На вход принимает текущий кадр и флаг работы программы. Если в вспомогательной программе появится ошибка, будет выведено соответствующее сообщение и флаг работы программы получит значение `false`.

```

1 void GMOD::start(cv::Mat& camera_frame_raw,
2                   bool& _load_flag) {
3     auto status = this->_runMPPGraph(camera_frame_raw, _load_flag);
4     if (!status.ok()) {
5         std::string msg(status.message());
6         LOG(ERROR) << (msg);
7         _load_flag = false;
8     }
9 }
```

Листинг 4.7: Функция `start()`

Вспомогательная функция `absl::Status _initMPPGraph(cv::VideoCapture& capture)`, вызываемая в `init()`, которая инициализирует параметры камеры и графа конвейера. Изначально из proto-файла извлекается граф в виде `std::string`. Затем, при помощи стандартной функции MediaPipe, данный граф анализируется (парсится) и создается переменная `config`, отвечающая за структуру графа. Затем непосредственно создается новый граф в переменную `_graph`. Настраивается камера: устанавливаются значения заданные в `setCamera()`. Далее вспомогательные пакеты графа инициализируются значениями, заданными в `init()`. `Output_poller` привязывается к выходному потоку конвейера. Записывается изображение фона в `background`. Запускается граф. Если все выполнилось успешно, функция возвращает сообщение об этом.

```

1  absl::Status GMOD::_initMPPGraph(cv::VideoCapture& capture) {
2      std::string calculator_graph_config_contents;
3      MP_RETURN_IF_ERROR(mediapipe::file::GetContents(
4          graphFilename,
5          &calculator_graph_config_contents));
6
7      LOG(INFO) << "Get calculator graph config contents: "
8          << calculator_graph_config_contents;
9
10     mediapipe::CalculatorGraphConfig config =
11         mediapipe::ParseTextProtoOrDie<mediapipe::CalculatorGraphConfig>(
12             calculator_graph_config_contents);
13
14     LOG(INFO) << "Initialize the calculator graph.";
15     _graph.reset(new mediapipe::CalculatorGraph());
16     MP_RETURN_IF_ERROR(_graph->Initialize(config));
17
18     LOG(INFO) << "Initialize the camera";
19     capture.open(camId);
20     RET_CHECK(capture.isOpened());
21     capture.set(cv::CAP_PROP_FRAME_WIDTH, camResX);
22     capture.set(cv::CAP_PROP_FRAME_HEIGHT, camResY);
23     capture.set(cv::CAP_PROP_FPS, camFps);
24 }
```

```

25 std :: map<std :: string , mediapipe :: Packet> input_side_packets ;
26 input_side_packets [ "type_of_effect" ] = mediapipe :: MakePacket<int>(this->
27     typeOfEffect) ;
28 input_side_packets [ "model_selection" ] = mediapipe :: MakePacket<int>(this->
29     typeOfModel) ;
30
31 LOG(INFO) << "Start running the calculator graph." ;
32
33 auto output_poller_sop = _graph->AddOutputStreamPoller(kOutputStream) ;
34 RET_CHECK(output_poller_sop.ok()) ;
35 output_poller =
36     std :: make_unique<mediapipe :: OutputStreamPoller>(
37         std :: move(output_poller_sop.value()) ) ;
38
39 MP_RETURN_IF_ERROR(_graph->StartRun(input_side_packets)) ;
40 LOG(INFO) << "Start grabbing and processing frames." ;
41 this->background = cv :: imread("testData/" + this->backgroundFilename) ;
42 return :: mediapipe :: OkStatus() ;
43 }
```

Листинг 4.8: Функция `_initMPPGraph()`

Вспомогательная функция `absl::Status _runMPPGraph(cv::Mat& camera_frame_raw, bool& _load_flag)`, вызываемая в `start()`, которая запускает конвейер, передает ему данные и обрабатывает данные, извлеченные из него. Полученный текущий кадр переводится в формат BGR и преобразуется в горизонтальный формат. Далее создается уникальный указатель типа `ImageFrame`, который ссылается на текущий кадр. Также инициализируется метка времени. Это сделано для передачи кадра в конвейер. Такой же процесс повторяется и для изображения фона. Далее из графа забирается пакет, при помощи `output_poller`. Если этого не удалось сделать, флаг работы программы переводится в `false` и обработка завершается. И, если в `setOverlay()` установлен флаг `true`, заменяет текущий кадр на полученное изображение из пакета. Если все выполнилось успешно, возвращает сообщение об этом.

```

1 absl :: Status GMOD :: _runMPPGraph( cv :: Mat& camera_frame_raw ,
2                                         bool& _load_flag ) {
3     cv :: Mat camera_frame ;
4     cv :: cvtColor( camera_frame_raw , camera_frame , cv :: COLOR_BGR2RGB ) ;
5
6     cv :: flip( camera_frame , camera_frame , /* flipcode=HORIZONTAL */ 1 ) ;
7
8     auto input_frame = absl :: make_unique<mediapipe :: ImageFrame>(
9         mediapipe :: ImageFormat :: SRGB , camera_frame . cols , camera_frame . rows ,
10        mediapipe :: ImageFrame :: kDefaultAlignmentBoundary ) ;
11     cv :: Mat input_frame_mat = mediapipe :: formats :: MatView( input_frame . get () ) ;
12     camera_frame . copyTo( input_frame_mat ) ;
13
14     size_t frame_timestamp_us =
15         (double)cv :: getTickCount () / (double)cv :: getTickFrequency () * 1e6 ;
16     MP_RETURN_IF_ERROR(_graph->AddPacketToInputStream(
17         kInputStream , mediapipe :: Adopt( input_frame . release () )
18             . At( mediapipe :: Timestamp( frame_timestamp_us ) ) ) ) ;
19
20     auto background_frame = absl :: make_unique<mediapipe :: ImageFrame>(
```

```

21     mediapipe :: ImageFormat :: SRGB, background . cols , background . rows ,
22     mediapipe :: ImageFrame :: kDefaultAlignmentBoundary );
23     cv :: Mat background _ data = mediapipe :: formats :: MatView( background _ frame . get () );
24     background . copyTo( background _ data );
25
26     MP_RETURN_IF_ERROR( _ graph ->AddPacketToInputStream(
27         kInputBackground , mediapipe :: Adopt( background _ frame . release () )
28             . At( mediapipe :: Timestamp( frame _ timestamp _ us ))));
29
30     mediapipe :: Packet packet ;
31     if ( !output _ poller ->Next( &packet )) {
32         _ load _ flag = false ;
33         return absl :: Status () ;
34     }
35
36     auto& output _ frame = packet . Get <mediapipe :: ImageFrame>();
37     cv :: Mat output _ frame _ mat = mediapipe :: formats :: MatView( &output _ frame );
38     cv :: cvtColor( output _ frame _ mat , output _ frame _ mat , cv :: COLOR_RGB2BGR );
39
40     if ( showOverlay ) {
41         output _ frame _ mat . copyTo( camera _ frame _ raw );
42     }
43
44     return :: mediapipe :: OkStatus () ;
45 }

```

Листинг 4.9: Функция `_runMPPGraph()`

4.2 Создание проекта в среде Visual Studio и его настройка

При помощи системы сборки bazel собираем получившийся проект (классы, калькуляторы и конвейер), создавая при этом dll файл для импорта классов в Visual Studio. После создания проекта в Visual Studio необходимо его настроить. Для этого в настройках нужно указать путь к указать dll файл рис. 4.1 и к исходным файлам классов рис. 4.2.

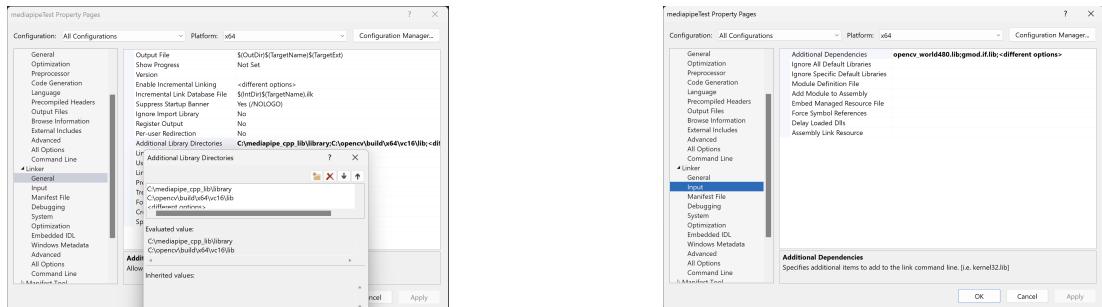


Рисунок 4.1 — Настройки Linker

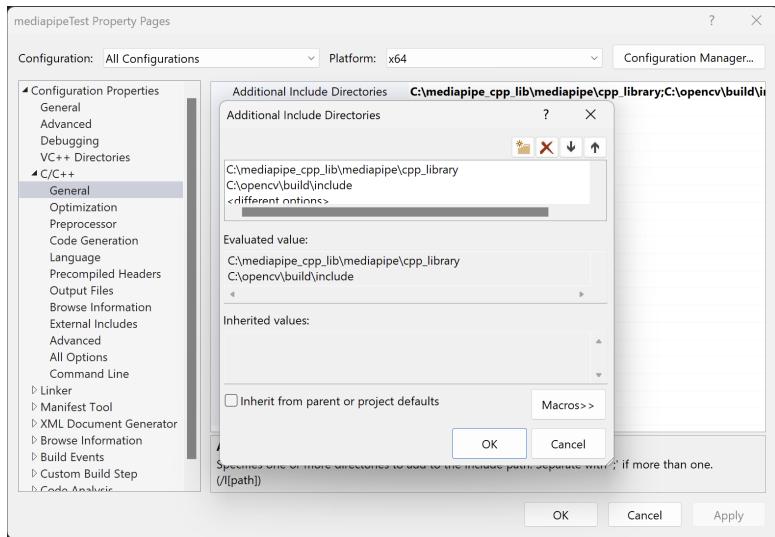


Рисунок 4.2 — Настройки C/C++ -> General

Все готово для запуска тестовой программы:

```

1 #include <iostream>
2 #include <typeinfo>
3 #include <opencv2/opencv.hpp>
4
5 #include "gmod_api.h"
6
7 int main() {
8     IGMOD* test = CreateGMOD();
9     test->setCameraProps(0, 1280, 720, 30);
10    test->setOverlay(true);
11
12    cv::VideoCapture cap;
13    test->init("graph.pbtxt", cap);
14
15    int frames = 0;
16    double fps = 0.0;
17    std::string path;
18    bool load_flag = true;
19    auto start_time = std::chrono::high_resolution_clock::now();
20
21    while (load_flag) {
22        cv::Mat temp;
23        cap >> temp;
24        test->start(temp, load_flag);
25
26        frames++;
27        auto end_time = std::chrono::high_resolution_clock::now();
28        double elapsed_time = std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time).count() / 1000.0;
29        if (elapsed_time >= 1.0) {
30            fps = frames / elapsed_time;
31            frames = 0;
32            start_time = std::chrono::high_resolution_clock::now();
33        }
34
35        std::stringstream ss;
36        ss << "FPS: " << std::fixed << std::setprecision(1) << fps;
37        cv::putText(temp, ss.str(), cv::Point(10, 30), cv::FONT_HERSHEY_SIMPLEX,
38                    1, cv::Scalar(0, 0, 255), 2);

```

```

38     cv::imshow("test", temp);
39     const int pressed_key = cv::waitKey(5);
40     if (pressed_key == 27) load_flag = false;
41     if (cv::getWindowProperty("test", cv::WND_PROP_VISIBLE) < 1) load_flag =
42         false;
43     if (pressed_key == 'w') {
44         int type;
45         std::cin >> type;
46         path = "background.jpg";
47         if (type == 0) {
48             test->setOverlay(false);
49             test->init("mediapipe_graphs/selfie_segmentation/ultimate.pbtxt"
50                         , cap, path, 0);
51         } else if (type == 1) {
52             test->setOverlay(true);
53             test->init("mediapipe_graphs/selfie_segmentation/ultimate.pbtxt"
54                         , cap, path, 1);
55         }
56     }
57 }

```

Листинг 4.10: Основная программа в Visual Studio

Результат работы программы:



Рисунок 4.3 — Результат работы программы

Заключение

В ходе проекта:

1. Была рассмотрена задача обработки 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. Рассмотрена архитектура конвейера и реализованы необходимые калькуляторы для него;
5. Реализованы классы для работы с фреймфорком MediaPipe в среде разработки Visual Studio;
6. Успешно протестированы результаты работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Google. (2023). MediaPipe Framework [Electronic resource]: Google for Developers. Режим доступа: <https://developers.google.com/mediapipe/framework>. Дата доступа: 12.12.2023
2. TensorFlow. (2023). Segmentation Lite. [Electronic resource]: TensorFlow Lite. Режим доступа: <https://www.tensorflow.org/lite/examples/segmentation/overview>. Дата доступа: 12.12.2023
3. Lugaresi, C., Tang, J., Nash, H., McClanahan, C., Uboweja, E., Hays, M., Zhang, F., Chang, C., Yong, M. G., Lee, J., Chang, W. T., Hua, W., Georg, M., Grundmann, M. (2019). MediaPipe: A Framework for Building Perception Pipelines [Electronic resource]. Режим доступа: <https://arxiv.org/pdf/1906.08172.pdf>. Дата доступа: 12.12.2023
4. bazel.build. Bazel [Electronic resource]. Режим доступа: <https://bazel.build>. Дата доступа: 12.12.2023