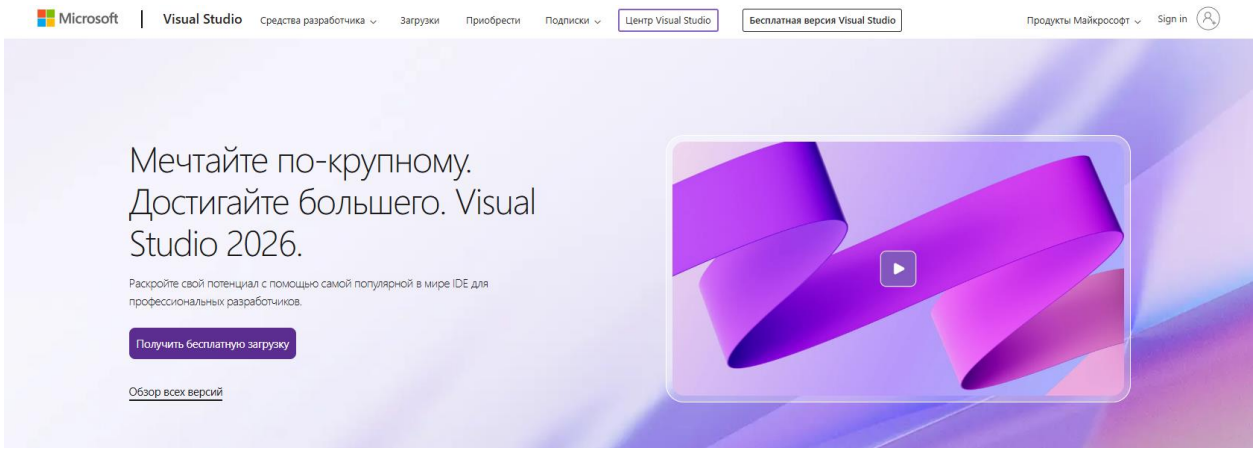
Работа с библиотекой OpenCV на языке C++

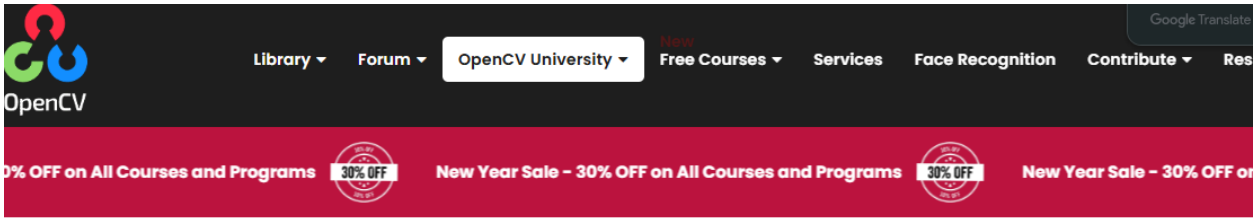
Установка Visual Studio и библиотеки OpenCV

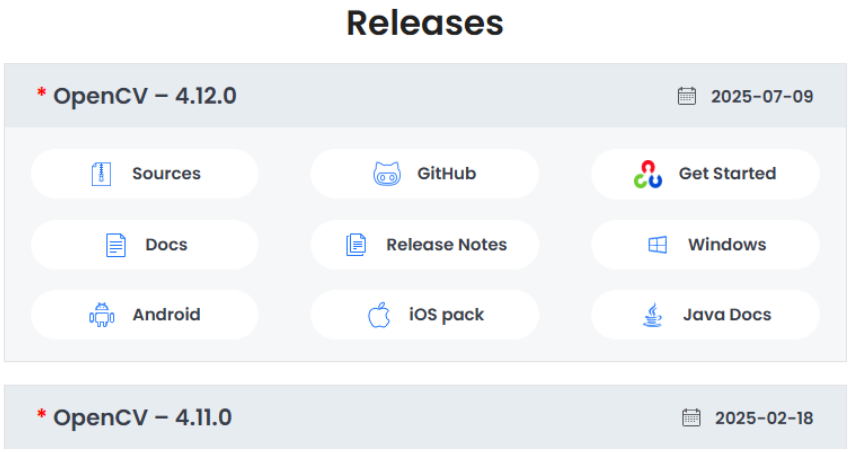
Для начала нужно скачать два файла:

1) Visual Studio с сайта visualstudio.microsoft.com/ru/

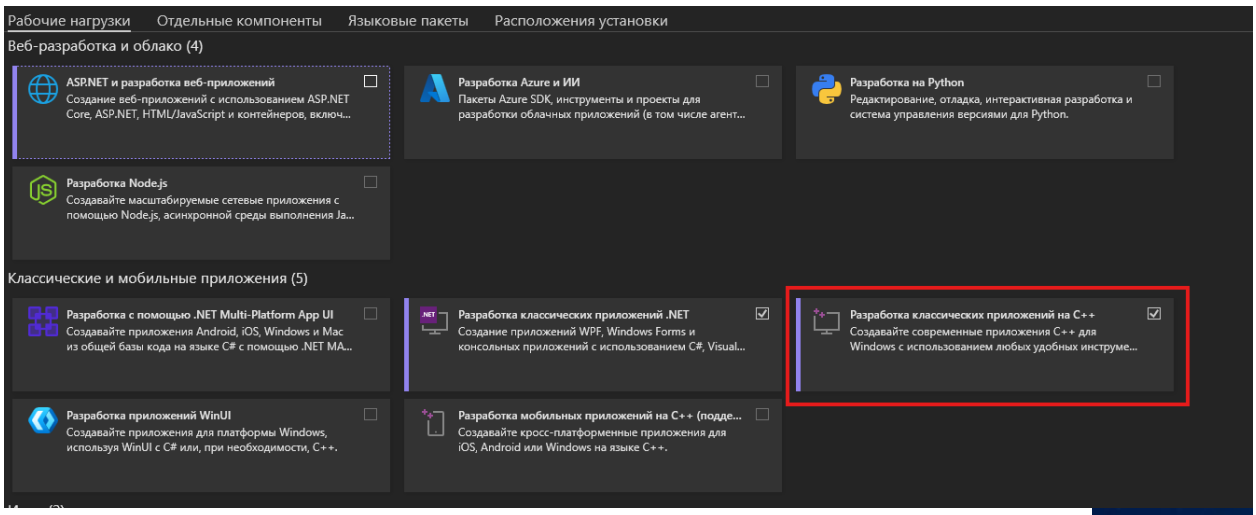


2) OpenCV последней версии для Windows с сайта opencv.org/releases/

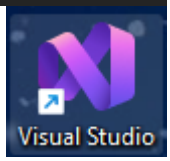




Сохраняем их в любую удобную папку. Запускаем установочный файл Visual Studio. При установке ***обязательно*** выбираем пункт «**Разработка классических приложений на C++».** Если Visual Studio у вас уже установлен, но без этого пункта, запустите через поиск приложение **Visual Studio Installer**, нажмите **Изменить** и выберите нужный пункт.



По окончанию установки на рабочем столе появится ярлык. Пока не трогаем его.

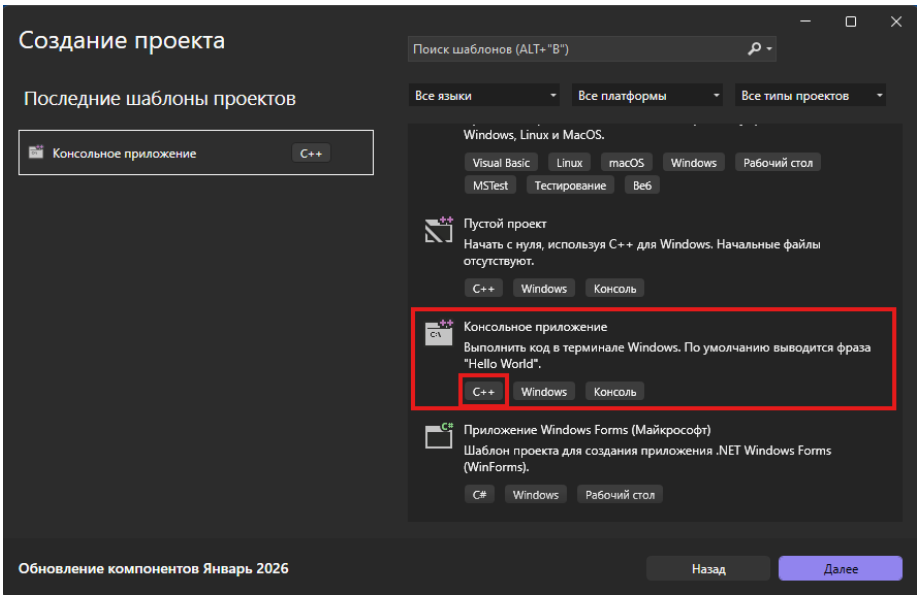


Запускаем скаченный файл **opencv-4.12.0-windows.exe** (или более новая версия). В данном случае установка – это обычная распаковка. Распаковываем туда, откуда будет потом удобно копировать и оставляем открытым.

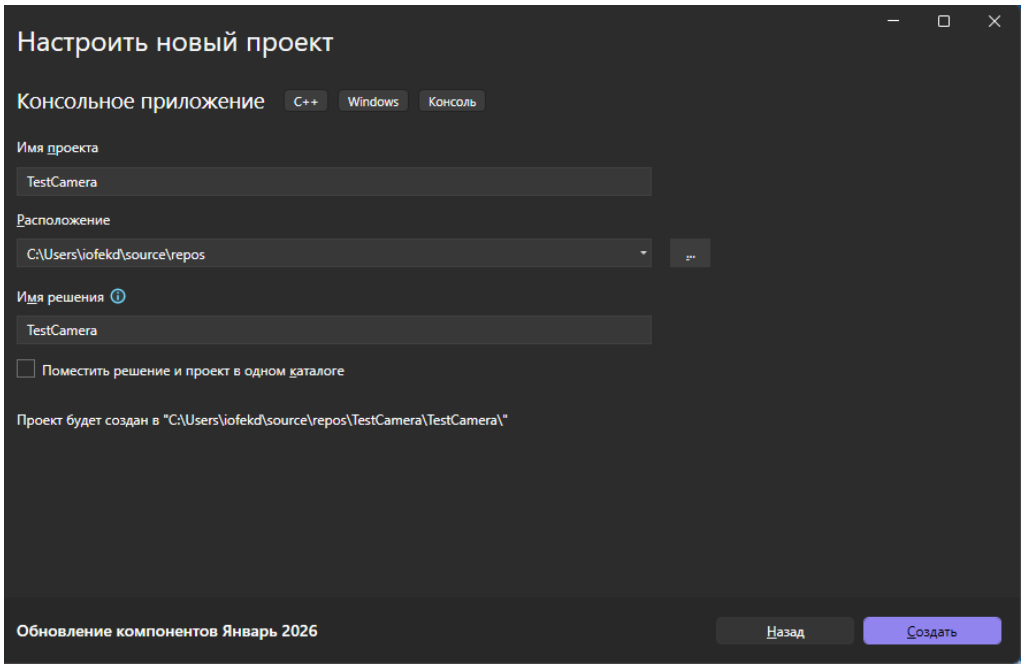
Создание и настройка проекта

Закрываем папку и открываем **Visual Studio** при помощи созданного ярлыка.

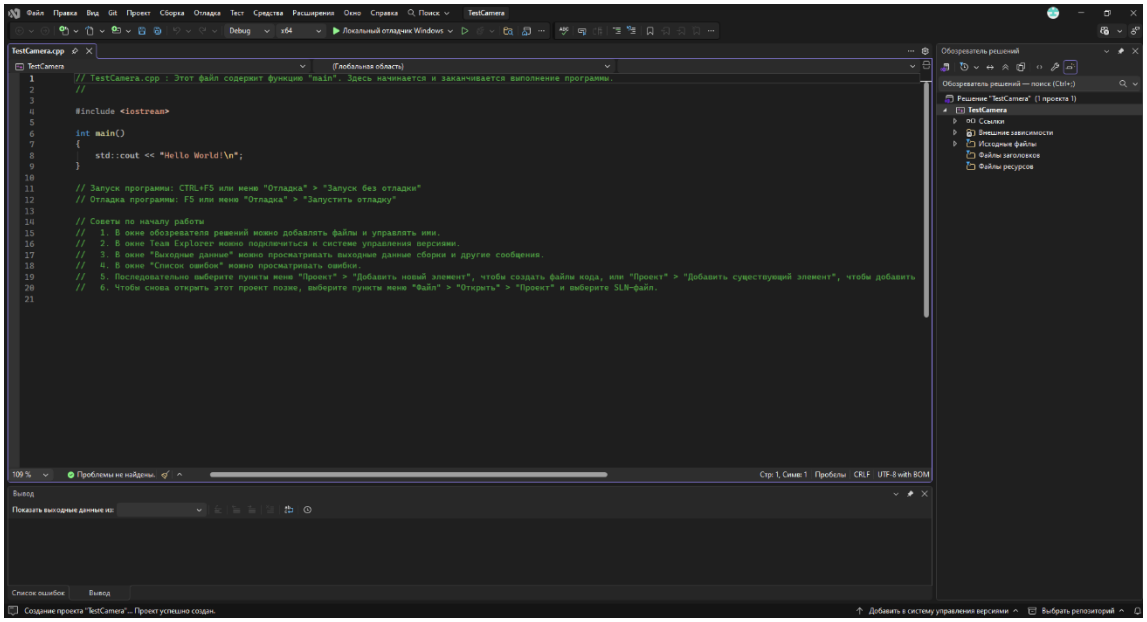
Видим прошлые работы (пока их нет, но в дальнейшем будет удобно открывать). Нажимаем Создание проекта. Предлагается выбрать тип создаваемого



приложения. Выбираем Консольное приложение C++. Важно, Консольное приложение может быть не одно. Требуется именно C++!
На следующей странице нужно ввести данные о проекте. Пишем Имя проекта, указываем расположение. Если поставить галочку «Поместить решение и проект в одном каталоге», то для программы не будет создана отдельная папка. Не рекомендую это делать. Когда все готово, нажимаем **Создать**.

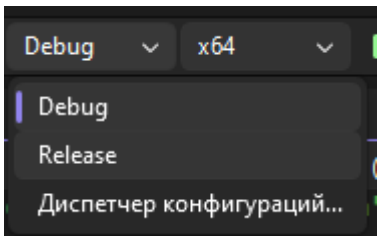


Открывается окно с проектом. Основное окно с программой, снизу лог и вывода данных, справа древо файлов проекта. Но прежде, чем начинать программировать, нужно настроить среду.



Первое, что требуется – переключиться с режима Debug на Release. Или можно этого не делать, тогда программа будет работать медленнее, но будет возможность к более точной отладке и настройке. Краткая разница между ними:

Debug (Отладка)



Можно:

- ставить брейкпоинты
- идти по строкам (F10 / F11)
- смотреть значения переменных
- без оптимизаций
- код компилируется почти «как написан»

Минусы:

- медленный
- может работать иначе, чем Release
- не предназначен для конечного пользователя

Release (Выпуск)

Оптимизация кода: убирает лишние переменные, разворачивает циклы, инлайнит функции

Нет отладочной информации: брейкпоинты не работают, шагать по строкам нельзя

Плюсы:

- Меньше проверок
- EXE меньше
- Работает быстрее (иногда в разы)

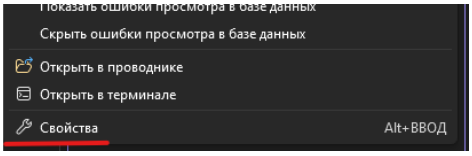
Минусы: Ошибки типа:

- неинициализированные переменные
- гонки
- UB (undefined behavior)

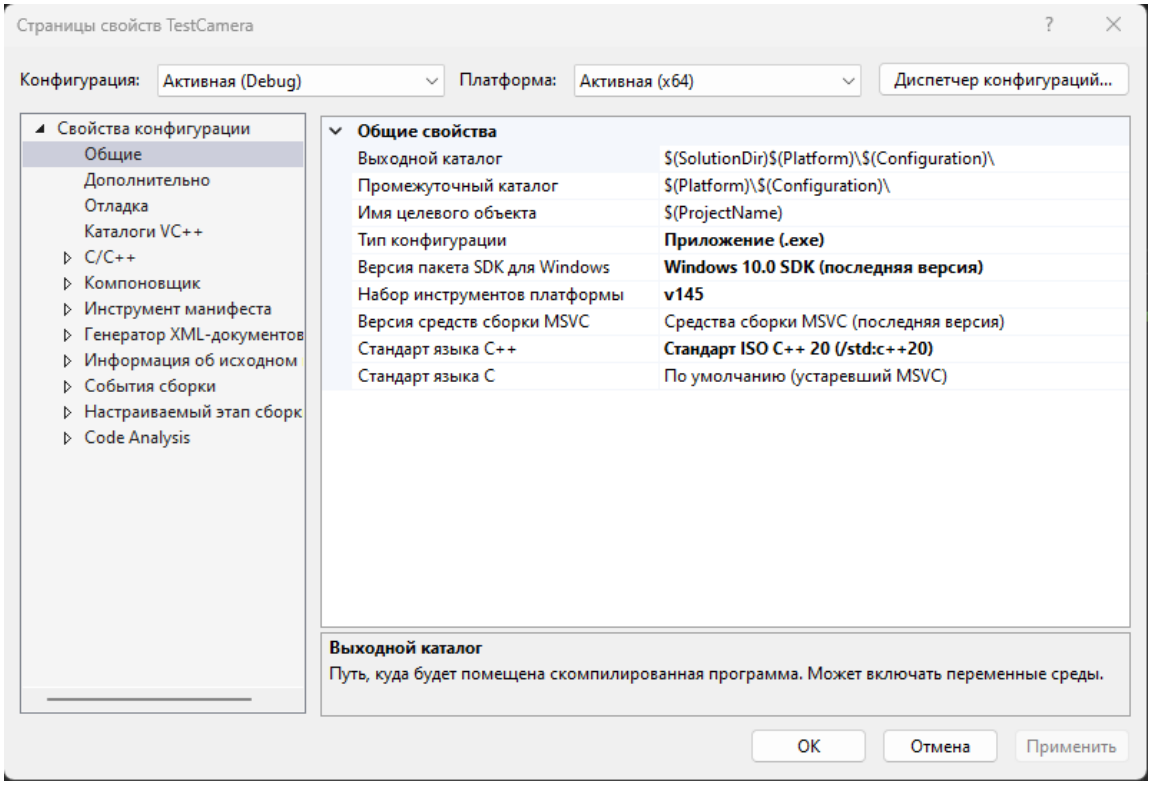
Я работаю сразу с Release, но если вы хотите сначала использовать Debug, то все действия, указанные ниже нужно проделать дважды, для каждого из режимов.

Теперь нужно перенести распакованные файлы библиотеки в ваш проект. Находим распакованную папку opencv и копируем её в проект рядом с файлом **ИмяПроекта.slnx**. Весит папка много, но часть файлов из нее можно удалить. Нам нужна только **Build**, а в ней только **include** и **x64**.

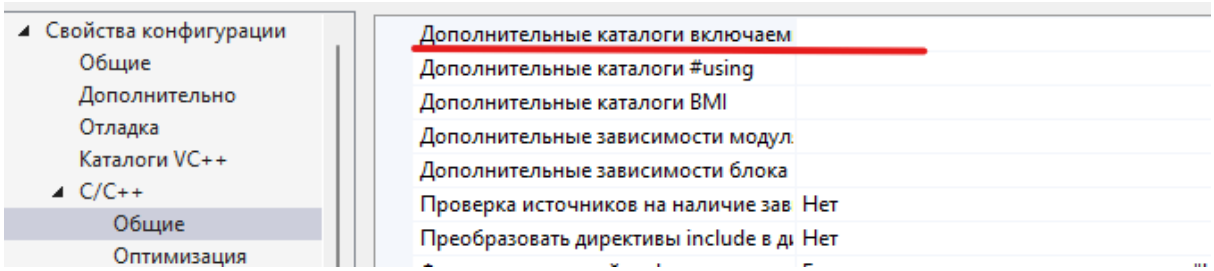
Возвращаемся в Visual Studio, выбираем режим (**Debug** или **Release**), нажимаем на имя проекта в дереве справа правой кнопкой мыши, выбираем пункт **Свойства.**



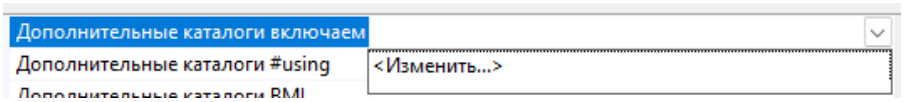
Открываются настройки проекта:



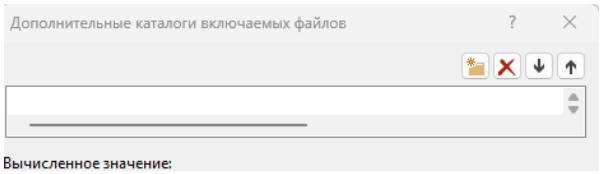
Для начала нас интересует вкладка **C/C++,** раздел **Общие**. В ней нужно будет поменять пункт «**Дополнительные каталоги включаемых файлов**»



Для их изменения нажимаем на них, появляется кнопка <Изменить…>.

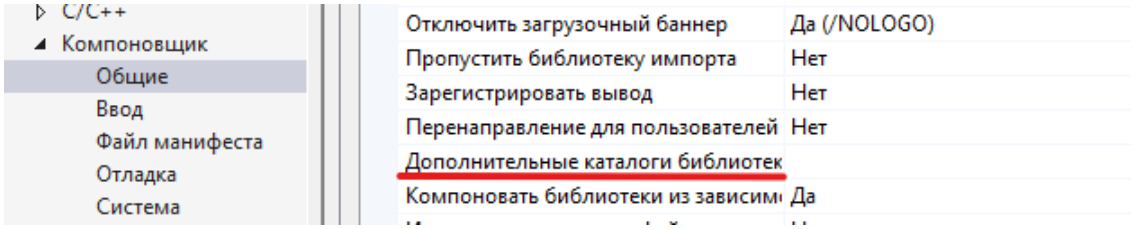


При нажатии появляется окно:



Вводим значение: **$(SolutionDir)opencv\build\include** и нажимаем ОК.

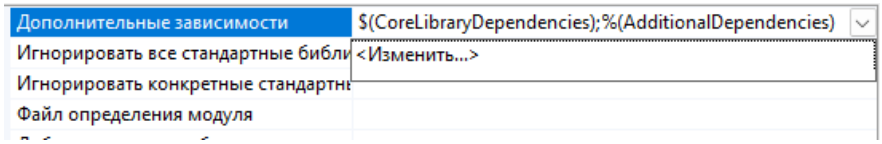
После этого открываем вкладку **Компоновщик**, раздел **Общие**. Тут изменяем пункт «**Дополнительные каталоги библиотек**».



Вводим значение: **$(SolutionDir)opencv\build\x64\vc16\lib** и нажимаем ОК.

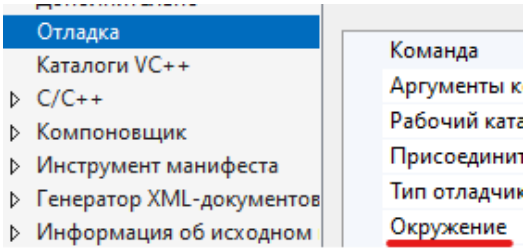
После этого заходим в этой же вкладке **Компоновщик** находим раздел **Ввод**.

Нас интересует только первая строка «**Дополнительные зависимости**»



Нажимаем **Изменить** и вводим **opencv_world4120.lib** для **Release** или **opencv_world4120d.lib** для **Debug**.

Нажимаем **ОК**.



Далее открываем раздел Отладка, пункт Окружение и вводим в него:

PATH=$(SolutionDir)opencv\build\x64\vc16\bin;%PATH%

Нажимаем **ОК**.

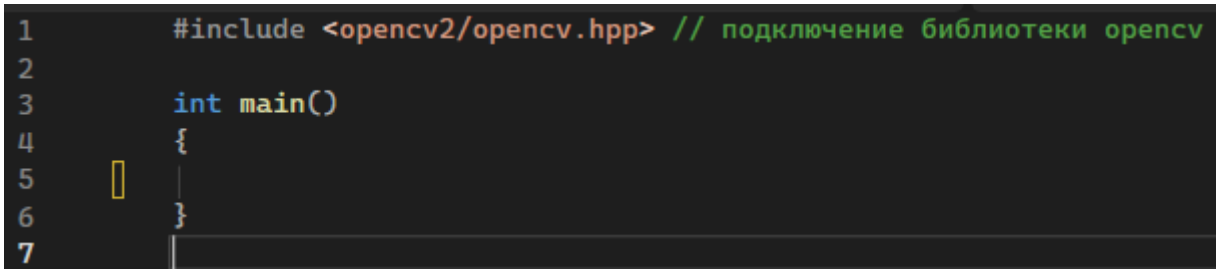
Всё, программа готова к работе. Для проверки стоит написать в 1 строке:

```
#include <opencv2/opencv.hpp>
```

Если система не выдает ошибок, значит библиотека подключена верно и можно начинать работу с камерой.

Запускаем трансляцию с камеры

Для начала работы напишем простейший код для вывода кадров с камеры в окно на экране. Для этого нужно удалить весь код, что изначально был в системе, кроме int main() { } и созданной нами строки с подключением библиотеки.



Далее есть 2 варианта. Можно создать пространство имен для OpenCV и в дальнейшем не писать каждый раз «**cv::**» перед всеми командами из библиотеки. Делается это так:

```
using namespace cv;
```

После этого команды будут писаться так: «`Mat frame`», вместо «`cv::Mat frame`»

Но это считается не правильным, хоть и упрощает написание, поэтому в примерах я так писать не буду.

Весь дальнейший код будет в `int main()`.

Для начала нужно создать два объекта классов.

1) Класс Mat – это контейнер для изображения. Если совсем просто**:** `cv::Mat` = «умная матрица пикселей»**.**
2) VideoCapture — это класс OpenCV для получения видео.

```
cv::Mat frame;
cv::VideoCapture video(0, cv::CAP_DSHOW);
```

В объекте `frame` (рамка) будет хранится полученное изображение. Объект `video` будет получать кадры с камеры. У него есть два параметра. Первый - № подключённой камеры. Чаще всего это 0 или 1. Встроенная камера чаще под № 0. Второй – указание, что использовать OpenCV для работы с камерой. (`CAP_DSHOW` - это DirectShow, мультимедийный API Windows).

После этого стоит добавить строку защиты, чтобы, если камера не открылась, система сама закрыла программу и выдала код ошибки, явно указывающий на то, в чём проблема:

```
if (!video.isOpened()) return -1; // если камера не открылась – заверши программу с кодом ошибки -1
```

Это подготовительный этап программы, настраивающий оборудование. В дальнейшем он может занимать сотни строк, когда будет множество устройств, требующих инициализации и настройки.

Далее создается цикл (конечный с условием остановки или бесконечный, тогда остановка создается условием внутри цикла):

```
while (true) // бесконечно
{
    // здесь будет код
    if (cv::waitKey(1) == 27) break; // ожидание 1 мс и проверка нажатия клавиши ESC
}
```

Строка с условием выполняет сразу три важных действия – добавляет в цикл задержку в 1 мс, даёт OpenCV обработать события в окне и проверяет нажатие клавиши остановки. 27 – это клавиша ESC на клавиатуре.

Остается самое главное – получить картинку с камеры и вывести её. Для получения изображения используем наши объекты:

```
video >> frame;
```

Передаём картинку из видеопотока в матрицу изображения.

```
if (frame.empty()) break;
```

Проверяем. Не пришёл ли пустой кадр. Если да, значит камера отключилась – завершаем программу.

```
imshow("OriginalVideo", frame);
```

`imshow` – функция, создающая окно с указанным названием и выводящая в него то, что записано во 2 параметре. В нашем случае – кадр.

В итоге получается следующая программа:

```
#include <opencv2/opencv.hpp> // подключение библиотеки opencv

int main()
{
    cv::Mat frame; // объект класса Mat для хранения текущего кадра изображения
    cv::VideoCapture video(0, cv::CAP_DSHOW); // создаем объект класса VideoCapture: 0 – индекс камеры, CAP_DSHOW – backend DirectShow
    if (!video.isOpened()) return -1; // если камера не открылась – завершаем программу с кодом ошибки

    while (true) // бесконечно
    {
        video >> frame; // получаем кадр из видеопотока
        if (frame.empty()) break; // если кадр пустой – выходим из цикла
        imshow("OriginalVideo", frame); // отображаем кадр в окне с именем OriginalVideo
        if (cv::waitKey(1) == 27) break; // ожидание 1 мс и проверка нажатия клавиши ESC
    }
}
```

Обработка изображения

Немного о цветовых режимах

Цвет на изображении — это не «просто цвет», а набор чисел, который интерпретируется по определённым правилам. Эти правила называются цветовой моделью или цветовым пространством. От выбранной модели напрямую зависит, насколько удобно и точно можно выделять объекты по цвету.

Разберем четыре модели: **RGB**, **BGR**, **HSV** и **LAB**.

RGB (Red, Green, Blue)

Модель **RGB** описывает цвет как сумму трёх компонентов:

- **R** — интенсивность красного,
- **G** — зелёного,
- **B** — синего.

Каждый пиксель хранит три числа (обычно от 0 до 255), например:

R = 255, G = 0, B = 0 → чисто красный

R = 255, G = 255, B = 255 → белый

R = 0, G = 0, B = 0 → чёрный

RGB — это модель для отображения, а не для анализа.

Она хорошо подходит для экранов, но плохо подходит для поиска цвета**,** потому что:

- яркость и цвет сильно смешаны,
- один и тот же цвет при разном освещении даёт совсем другие значения R, G и B.

BGR (Blue, Green, Red)

OpenCV по умолчанию использует BGR, а не RGB. Это не ошибка и не «особенность камеры» — это историческое решение OpenCV.

Важно помнить:

- `cv::imshow()` ожидает изображение именно в BGR,

- большинство функций OpenCV работают с BGR напрямую,
- если перепутать порядок каналов, цвета будут выглядеть неправильно.

BGR/RGB используются в основном для вывода изображения**,** но редко — для анализа цвета.

HSV (Hue, Saturation, Value)

HSV — одна из самых удобных моделей для поиска объектов по цвету**.**

Она описывает цвет так, как его воспринимает человек:

- **H (Hue)** — цветовой тон (красный, зелёный, синий и т.д.)
- **S (Saturation)** — насыщенность (насколько цвет “чистый” или серый)
- **V (Value)** — яркость

В OpenCV:

- **H** ∈ [0, 179]
- **S** ∈ [0, 255]
- **V** ∈ [0, 255]

Главное преимущество **HSV**: Цвет отделён от яркости

Это означает, что при изменении освещения Hue остаётся примерно тем же**,** объект легче выделить даже при тенях или бликах.

Пример: красный мяч остаётся красным и на свету, и в полутени, в **RGB** его значения сильно меняются, в **HSV** меняется в основном **V**, а **H** остаётся стабильным.

Именно поэтому для задач: поиска объектов, трекинга, сегментации по цвету чаще всего используют **HSV**.

LAB (CIELAB)

LAB — более сложная, но очень мощная цветовая модель.

Она состоит из трёх каналов:

- **L** — светлота (Lightness),
- **A** — ось от зелёного к красному,

- **B** — ось от синего к жёлтому.

Особенности **LAB**:

- цвет и яркость разделены **ещё сильнее**, чем в **HSV**,
- изменения освещения влияют в основном на **L**,
- **A** и **B** остаются более стабильными.

LAB считается перцептивно равномерной моделью**:** одинаковое числовое изменение ≈ одинаковое визуальное изменение

Это делает **LAB** особенно полезной:

- при сложном освещении,
- при поиске объектов похожих по цвету,
- когда **HSV** даёт слишком много шумов.

Минусы **LAB**:

- сложнее для понимания,
- диапазоны значений менее интуитивны,
- работает медленнее, чем **HSV**.

Поиск объектов по цвету

Когда изображение получено и выведено, его можно обрабатывать.

Нашей задачей является поиск объекта по цвету. Для того, чтобы это сделать, нужно выполнить следующие действия:

1) Преобразовать полученную картинку из BGR в другие цветовые системы.
2) Найти все объекты указанного цвета и построить бинарную маску найденных объектов.
3) Выбрать наибольший из объектов.
4) Построить окружность вокруг крайних точек объекта и найти её центр.

1. **Преобразование модели**

Для преобразования картинки в другую цветовую систему используется команда `cv::cvtColor(frame, frame2, cv::COLOR_BGR2HSV);`

`frame` – объект типа `Mat`, содержащий входное изображение;

`frame2` – объект типа `Mat`, в который будет добавлено выходное изображение;

`cv::COLOR_BGR2HSV,` параметр, указывающий какое преобразование будет осуществлено (например, BGR в HSV).

После выполнения этого преобразования можно вывести оба изображения и увидеть разницу между двумя цветовыми режимами. Для этого можно создать два отдельных окна:

```
imshow("OriginalVideo", frame); // отображаем кадр в окне с именем OriginalVideo
imshow("HSVVideo", frame2); // отображаем кадр в окне с именем HSVVideo
```

Попробуйте преобразовать изображение в RGB, HSV и LAB и одновременно вывести в 4 разных окнах, чтобы увидеть разницу. *Не забудьте создать дополнительные объекты для каждого из изображений.*

Так как при поиске объекта по цвету нам не требуется большая резкость кадра, то стоит немного размыть полученные изображения. Это уберет некоторую часть «шума» и потом будет проще обработать кадр. Для этого используется команда

```
cv::GaussianBlur(frame2,
                 frame2,
                 cv::Size(15, 15),
                 0);
```

`frame2` – входное изображение;

`frame2` – выходное изображение (перерисовываем на том же кадре);

`cv::Size(15, 15)` – сила размытия в пикселях;

`0` – $\sigma \approx$ (kernel_size - 1) / 6;

2. Поиск объекта методом HSV

Для поиска вхождений используется команда.

```
cv::inRange(frame2,
            cv::Scalar(Hmin, Smin, Vmin),
            cv::Scalar(Hmax, 255, 255),
            frame2);
```

`frame2` – входное изображение;

`cv::Scalar(Hmin, Smin, Vmin),` – объединение параметров нижней границы по HSV. В данном случае задаются в переменных;

`cv::Scalar(Hmax, 255, 255),` – объединение параметров верхней границы по HSV. В данном случае только **H** задается в виде переменной, тк остальные максимальны.

`frame2` – выходное изображение (перерисовываем на том же кадре);

inRange преобразует наше изображение в чёрно-белый (бинарный) формат. Всё, что входит в выбранный диапазон становится белым, а всё, что не входит, чёрным. Таким образом можно находить объекты.

3. Поиск объекта методом Lab

Начать поиска объекта при помощи метода Lab можно так же, как в случае с HSV – преобразованием кадра командой `cvtColor`, но с параметром `COLOR_BGR2Lab:`

```
cv::cvtColor(frameOrig, frameLab, cv::COLOR_BGR2Lab);
```

Выходной параметр уже не `frame3`, а `frameLab` (не забудьте создать его). Остальные тоже переименуем, чтобы не запутаться. `frame1` назовём `frameOrig`, а `frame2 - maskHSV`(мы ищем именно маску, а не рамку).

После этого начинаются отличия. Изображение в Lab делится на три канала, но в поиске используются только A и B. Разделим их это методом `cv::split`.

Создадим динамический массив для хранения каналов изображения:

```
std::vector<cv::Mat> vectorFrameLab;
```

Важно, для работы динамических массивов в начале программы должна быть подключена библиотека `vector.`

Так же создадим две новые переменные типа `Mat` для хранения бинарных масок после поиска объектов:

```
cv::Mat maskA, maskB;
```

После этого разделяем изображение на каналы:

```
cv::split(frameLab, vectorFrameLab);
```

И ищем вхождения нужного цвета на каждой из них:

```
cv::inRange(vectorFrameLab[1], Amin, Amax, maskA);
cv::inRange(vectorFrameLab[2], Bmin, Bmax, maskB);
```

`vectorFrameLab[1]/[2]` – входное изображение

`Amin/Bmin` – нижняя граница поиска

`Amax/Bmax` – верхняя граница поиска

`maskA/maskB` – выходное изображение

Не забудьте создать переменные для указания диапазона поиска. Самый правильный способ — вывести изображение в Lab и посмотреть значения пикселя напрямую через setMouseCallback. Для этого стоит создать отдельную программу, чтобы не засорять текущую. Пример такой программы ниже, в конце раздела.

Создаем ещё одну переменную типа `Mat maskLab` и объединяем в нее оба полученных вхождения:

```
maskLab = maskA & maskB;
```

Маска Lab намного лучше, чем HSV, видит засвеченную картинку, тк почти не реагирует на освещение (до определенных пределов, конечно). Но самым эффективным способом поиска объекта является их объединение. Делается это просто: создаем новую переменную и указываем, что она равна HSV или LAB:

```
maskLabHSV = maskHSV | maskLAB;
```

Пример программы для чтения показаний Lab и HSV:

```
#include <opencv2/opencv.hpp>   // Подключение основной библиотеки OpenCV
#include <iostream>             // Подключение библиотеки для вывода в консоль

// Глобальные переменные для хранения изображений
cv::Mat frame;      // Кадр с камеры в формате BGR
cv::Mat frameLab;   // Тот же кадр, но преобразованный в цветовое пространство Lab
cv::Mat frameHSV;   // Тот же кадр, но преобразованный в цветовое пространство HSV

// Функция-обработчик событий мыши
// Она автоматически вызывается OpenCV при клике в окне
void onMouse(int event, int x, int y, int, void*)
{
    if (event == cv::EVENT_LBUTTONDOWN)// Проверяем, нажата ли левая кнопка мыши
    {
        // Получаем значение пикселя в координатах (x, y)
        // Vec3b — это вектор из трёх байтов
        cv::Vec3b pixelLAB = frameLab.at<cv::Vec3b>(y, x);
        cv::Vec3b pixelHSV = frameHSV.at<cv::Vec3b>(y, x);

        // Извлекаем значения каналов
        int L = pixelLAB[0];   // Канал яркости (Lightness)
        int A = pixelLAB[1];   // Канал зелёный ↔ красный
        int B = pixelLAB[2];   // Канал синий ↔ жёлтый

        int H = pixelHSV[0];   // Канал Hue
        int S = pixelHSV[1];   // Канал Saturation
        int V = pixelHSV[2];   // Канал Value

        // Выводим значения каналов в консоль
        std::cout << "L: " << L
            << "  A: " << A
            << "  B: " << B
            << std::endl;
        std::cout << "H: " << H
            << "  S: " << S
            << "  V: " << V
            << std::endl;
    }
}

int main()
{

    cv::VideoCapture cap(0); // Создаём объект для захвата видео с камеры
    if (!cap.isOpened())// Проверяем, удалось ли открыть камеру
        return -1;  // Если нет — завершаем программу
    cv::namedWindow("LabHSV");// Создаём окно с именем "LabHSV"
    cv::setMouseCallback("LabHSV", onMouse);// Назначаем функцию обработки клика
мыши для окна "LabHSV"
    while (true)// Бесконечный цикл захвата видео
    {
        cap >> frame;// Захватываем новый кадр с камеры
        cv::cvtColor(frame, frameLab, cv::COLOR_BGR2Lab);// Преобразуем BGR в Lab
        cv::cvtColor(frame, frameHSV, cv::COLOR_BGR2HSV);// Преобразуем BGR в HSV
        cv::imshow("LabHSV", frame);// Отображаем исходное изображение (не LabHSV!)
        if (cv::waitKey(1) == 27)
            break;                // Если нажали ESC — выходим из цикла
    }
    return 0;  // Завершение программы
}
```

4. Поиск контуров объектов

Теперь, когда есть итоговая бинарная маска `maskLabHSV`, то мы можем начать искать на ней нужные нам объекты, чтобы в дальнейшем с ними взаимодействовать. Для этого нужно отсеять лишнее и выбрать только самый крупный объект.

Создадим двухмерный массив координат, чтобы записывать в него контуры объектов:

```
std::vector<std::vector<cv::Point>> contour;
```

После этого нужно воспользоваться функцией:

```
cv::findContours(
    frameLabHSV,
    contour,
    cv::RETR_EXTERNAL,
    cv::CHAIN_APPROX_SIMPLE);
```

Эта функция извлекает из бинарной маски все найденные контуры по указанным параметрам:

`frameLabHSV` – входные данные

`contour` – выходные данные

`cv::RETR_EXTERNAL` – режим работы:

`RETR_EXTERNAL` – берет только внешние контуры;

`RETR_TREE` – внутренние дыры станут отдельными контурами

`cv::CHAIN_APPROX_SIMPLE` – режим обработки контура:

`CHAIN_APPROX_SIMPLE` – упрощение контура;

`CHAIN_APPROX_NONE` – обработка каждого пикселя без упрощения.

Выполнение этой функции с указанными параметрами позволит получить контуры всех объектов нужного цвета, найденных на кадре.

После этого нужно просто перебрать все объекты и найти среди них максимальный. Делается это очень просто:

Создаем переменные для хранения максимального контура и его индекса:

```
double maxArea = 0
int maxIndex = -1;
```

И просматриваем в цикле каждый контур, не больше ли он чем максимальный. Если да – записываем его, как наибольший, и запоминаем индекс.

```
for (int n = 0; n < contour.size(); n++)
{
    double area = cv::contourArea(contour[n]);
    if (area > maxArea)
    {
        maxArea = area;
        maxIndex = n;
    }
}
```

5. Выделение наибольшего контура и указание его центра

Когда найден наибольший контур, хочется понять, является ли он искомым объектом, и, если да, обвести его в кружок на основном кадре. Для этого нужно узнать сравнить его размер с минимальным порогом (и заодно проверить, есть ли вообще объект):

```
if (maxIndex != -1 && maxArea > 200)
```

Если объект найден и больше 200 точек, то с ним можно работать.

Создадим переменные для определения радиуса круга, которым обведем объект и точки его центра:

```
cv::Point2f objectCenter;
float objectRadius = 0;
```

После этого можно рассчитать окружность. Используем функцию:

```
cv::minEnclosingCircle(contour[maxIndex], objectCenter, objectRadius);
```

Проверяем, правда ли полученная окружность достаточного размера, чтобы считаться нормальным объектом. Если да, то рисуем её и маленький кружок в точке центра. Обратите внимание, что рисуем мы на изначальном кадре, том, что мы получили с камеры, не на масках.

```
if (objectRadius > 5)
{
  cv::circle(frameOrig, objectCenter, (int) objectRadius, cv::Scalar(0, 255, 0), 2);
  cv::circle(frameOrig, objectCenter, 3, cv::Scalar(0, 0, 255), -1);
}
```

Параметры команды рисования круга:

```
circle (Входное изображение, координаты центра, размер круга, цвет, толщина линии)
```

Создание трекбаров для настройки параметров

Для создания трекбаров используется функция

```
cv::createTrackbar("TrackName", "WinName", min, max);
```

Где `TrackName` – имя трекбара, `WinName` – имя окна, где будет находится трекбар, `min` и `max` – минимальное и максимальное значения для трекбара.

Чтобы создать трекбары, создадим отдельную функцию `initObjectControls()`. В ней создадим новое окно для трекбаров и зададим ему размер:

```
cv::namedWindow("Controls_Object", cv::WINDOW_NORMAL);
cv::resizeWindow("Controls_Object", 350, 260);
```

После создадим в этом окне трекбары для каждого из параметров:

```
cv::createTrackbar("H min", "Controls_Object", &H.min, 179);
cv::createTrackbar("H max", "Controls_Object", &H.max, 179);
```

… для остальных по аналогии, но максимальное значение будет 255.

Так как мы используем переменные `H.min`, `H.max`, `Smin` и другие, то придётся объявление этих переменных из локального `int main()` вынести в глобальную область, выше `initObjectControls()`. Структуру `struct MinMax` нужно перенести выше объявления этих переменных!

Вызываем созданную функцию в int main() до бесконечного цикла и получаем возможность регулировать параметры прямо из программы в отдельном окне.

Выравнивание изображения по координатам

При получении изображения с камеры оно может быть снято под углом — тогда прямоугольное поле выглядит как трапеция. В этом случае изображение можно выровнять (выполнить перспективное преобразование). Это особенно актуально, например, при соревновании Арканоид, когда нужно получить ровный прямоугольник поля по его углам.

Для начала создадим массив, в который будем заносить координаты углов:

```
std::vector<cv::Point2f> fieldCorners;
```

Далее создадим функцию, которая будет получать события мыши:

```
void onMouse(int event, int x, int y, int flags, void*)
```

Параметры функции:

`int event` – что произошло с мышью (код события):

EVENT_MOUSEMOVE // мышь движется
EVENT_LBUTTONDOWN // нажата левая кнопка
EVENT_LBUTTONUP // отпущена левая кнопка
EVENT_RBUTTONDOWN // нажата правая кнопка
EVENT_RBUTTONUP // отпущена правая кнопка
EVENT_MBUTTONDOWN // средняя кнопка
EVENT_MOUSEWHEEL // прокрутка колеса

`int x, int y` – координаты

`flags` –показывает, какие кнопки мыши или клавиши (*Shift*/*Ctrl*/*Alt*) зажаты в момент события.

`void*` – произвольный указатель на данные (можно передать самостоятельно при назначении callback).

Важно: эти параметры задаём не мы. Их передаёт OpenCV, когда вызывает нашу функцию.

Внутри функции проверяем два условия: нажата ли левая кнопка мыши и сколько точек уже сохранено. Если нажата ЛКМ и точек меньше четырёх — запоминаем координаты.

```
if (event == cv::EVENT_LBUTTONDOWN && fieldCorners.size() < 4)
    fieldCorners.emplace_back((float)x, (float)y);
```

Рядом с массивом точек зададим параметры, которые нам понадобятся:

Размеры поля после преобразования:

```
const int FIELD_W = 800;
const int FIELD_H = 400;
```

Статус калибровки (по умолчанию — не выполнена):

```
bool calibrated = false;
```

И матрица перспективного преобразования:

```
cv::Mat homography;
```

В `int main()` до цикла создаём окно для калибровки и назначаем обработчик мыши:

```
cv::namedWindow("View", cv::WINDOW_NORMAL);
cv::setMouseCallback("View", onMouse);
```

Также заранее добавим матрицу для уже выровненного изображения `frameTop` (там же, где в `int main()` создаются остальные `cv::Mat`).

В цикле будем проверять, произведена ли калибровка. Если нет — рисуем уже поставленные точки и ждём остальные. После того как все точки выбраны, ждём нажатия Enter и выполняем калибровку.

```
if (!calibrated)
```

В цикле `for` рисуем все уже полученные точки и выводим изображение:

```
for (size_t n = 0; n < fieldCorners.size(); n++)
    cv::circle(frameBGR, fieldCorners[n], 6, cv::Scalar(0, 0, 255), -1);

cv::imshow("View", frameBGR);
```

Проверяем нажатие клавиши (с задержкой 1 мс):

```
int k = cv::waitKey(1);
```

Далее проверяем: если нажата клавиша Enter и точек ровно 4, то считаем матрицу преобразования и закрываем окно:

```
if (k == 13 && fieldCorners.size() == 4)
{
    std::vector<cv::Point2f> dst = {
        {0,0},
        {(float)FIELD_W,0},
        {(float)FIELD_W,(float)FIELD_H},
        {0,(float)FIELD_H}
    };

    homography = cv::getPerspectiveTransform(fieldCorners, dst);
    calibrated = true;
    cv::destroyWindow("View");
}
```

Разберем подробно, что здесь происходит.

Массив `dst` — это четыре угла нового изображения (виртуального «поля»). Левый верхний угол — (0, 0), остальные задаются на основе размеров FIELD_W и FIELD_H (в нашем примере 800×400).

Функция `cv::getPerspectiveTransform(fieldCorners, dst)` вычисляет матрицу `homography`, которая переводит точки из исходного изображения (камеры) в координаты выровненного поля.

После этого мы ставим статус калибровки «выполнено» и закрываем окно калибровки.

В конце нашей калибровки нужно написать две важные строки:

```
if (k == 27) break;
continue;
```

Первая дублирует подобную строку из конца основного цикла, без неё невозможно будет завершить программу до окончания калибровки.

Вторая, не дает дальнейшей программе исполнятся, пока не завершена калибровка. Команда `continue` завершает текущую итерацию цикла и начинает новую, не исполняя дальнейший код, что позволяет выполнять основную программу только по окончанию калибровки.

Сразу после калибровки можно преобразовать изображение из `frameBGR` в `frameTop`:

```
cv::warpPerspective(frameBGR, frameTop, homography, cv::Size(FIELD_W, FIELD_H);
```

Важно: в дальнейшей программе замените `frameBGR` на `frameTop`, чтобы вся обработка выполнялась по уже выровненному изображению.

Обратите внимание: точки лучше ставить в одном и том же порядке — например, с левого верхнего угла по часовой стрелке. Если начать в другом порядке (например, с правого нижнего), можно получить перевёрнутое изображение — иногда это даже удобно (например, для Арканоида), но важно понимать, что это результат именно порядка точек.

Передача данных на ESP32 по UDP

Когда координаты объекта найдены, хочется их передать на робота, чтобы он мог среагировать на изменение положения объекта. Для этого можно использовать самые различные способы связи. Разберем, как это делать по WiFi при помощи протокола UDP.

Что такое UDP

UDP (User Datagram Protocol) — это сетевой протокол передачи данных, работающий поверх IP. В отличие от более сложного протокола TCP, UDP не устанавливает соединение между устройствами и не проверяет, дошли ли данные до получателя. Он просто отправляет пакет информации по указанному IP-адресу и порту — быстро и без дополнительных проверок. UDP работает по принципу «отправил и забыл». Передающая сторона формирует небольшой блок данных (датаграмму) и отправляет его в сеть. Принимающая сторона слушает определённый порт и, если пакет дошёл, обрабатывает его. Если пакет потерялся по дороге — протокол не пытается его переслать повторно.

Главное преимущество UDP — минимальная задержка.

Для задач управления роботом важнее получать актуальные данные быстро, чем гарантированно получать каждый пакет. Если один пакет с координатами потеряется, через несколько миллисекунд придёт следующий — уже более свежий. Именно из-за простоты и скорости UDP является удобным способом передавать координаты объекта на ESP32 по WiFi.

Подключение UDP

Для подключения UDP используется стандартная сетевая библиотека Winsock2. Помимо нее добавим ещё несколько библиотек для работы:

```
#include <winsock2.h> // основные сетевые типы и функции (сокеты)
#include <ws2tcpip.h> // удобные функции для IP-адресов и портов
#pragma comment(lib, "ws2_32.lib") // библиотека линковки WinSock.
```

После этого нужно создать два объекта – сокет для отправки данных и структуру для хранения IP и порта получателя:

```
SOCKET senderSocket = INVALID_SOCKET;  // сокет для отправки
sockaddr_in receiverAddr{};            // структура с IP и портом получателя
```

После создаются 3 функции. Первая инициализирует UDP в начале программы, вторая отключает его при завершении работы, а третья будет использоваться для отправки данных.

```
bool initUDP() // инициализация UDP-соединения
{
    WSADATA wsa; // структура для запуска WinSock
    // запускаем WinSock версии 2.2
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
        return false; // если ошибка – выходим
    // создаём UDP-сокет (IPv4, датаграммы, протокол UDP)
    senderSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    /* 1 параметр – тип адреса:
    * AF_INET = IPv4.
    * AF_INET6 → IPv6
    *
    * 2 параметр – тип сокета:
    * SOCK_DGRAM = датаграммы → это UDP.
    * SOCK_STREAM → TCP (соединение, подтверждения, контроль доставки)
    *
    * 3 параметр – Какой именно протокол использовать.
    * IPPROTO_UDP → UDP
    * IPPROTO_TCP → TCP */
    // проверяем, создан ли сокет
    if (senderSocket == INVALID_SOCKET)
        return false;

    receiverAddr.sin_family = AF_INET;      // используем IPv4
    receiverAddr.sin_port = htons(1919);     // порт получателя (перевод в сетевой
формат)

    // преобразуем строковый IP в бинарный формат
    inet_pton(AF_INET, "192.168.4.1", &receiverAddr.sin_addr);

    return true; // всё успешно
}


void shutdownUDP() // корректное завершение работы UDP
{
    // если сокет открыт – закрываем его
    if (senderSocket != INVALID_SOCKET)
        closesocket(senderSocket);

    senderSocket = INVALID_SOCKET; // помечаем как закрытый

    WSACleanup(); // освобождаем ресурсы WinSock
}


void sendData(const void* data, size_t size)
{
    sendto(
        senderSocket,                                // сокет
        reinterpret_cast<const char*>(data),         // указатель на данные
        static_cast<int>(size),                      // размер данных
        0,                                           // флаги
        reinterpret_cast<sockaddr*>(&receiverAddr),// адрес получателя
        sizeof(receiverAddr)                         // размер структуры адреса
    );
}
```

Передача данных

Протокол UDP передаёт данные не в виде «чисел» или «строк», а в виде последовательности байтов. Функция `sendto()` получает указатель на область памяти и количество байтов, которые нужно отправить. Всё остальное — это уже наша интерпретация этих байтов.

Это означает, что любые данные (координаты x и y, скорость, угол и т.д.) перед отправкой необходимо представить в виде массива байтов.

Как число хранится в памяти

Допустим, у нас есть координаты:

```
int x = 120;
int y = 85;
```

Компьютер хранит каждое число как набор байтов.

Например, 120 в шестнадцатеричном виде — `0x78`.

Если число занимает 4 байта, в памяти оно может выглядеть так:

```
0x78 0x00 0x00 0x00
```

UDP передаёт именно эти байты.

Важно понимать: сеть не знает, что это «координата». Это просто набор байтов. Принимающая сторона должна знать, как их правильно интерпретировать.

Передача координат в виде структуры

Для передачи координат x и y удобно использовать тип `uint16_t` (беззнаковое 16-битное число). Он занимает 2 байта и позволяет хранить значения от 0 до 65535, что более чем достаточно для координат изображения.

Создадим структуру для «упаковки» информации о координатах в пакет данных:

```
struct Packet
{
    uint16_t x;  // координата X
    uint16_t y;  // координата Y
};
```

Мы, по сути, создаем свой тип данных `Packet`, в который положим `x` и `y`.

```
Packet pack;
```

Теперь нужно подготовить координаты к передаче. Округлим их и ограничим диапазон, чтобы они не могли занимать более 4 байт.

```
int xi = (int)std::lround(objectCenter.x);
int yi = (int)std::lround(objectCenter.y);

xi = std::clamp(xi, 0, 65535);
yi = std::clamp(yi, 0, 65535);
```

`std::lround` – функция округления числа. В данном случае мы округляем координату, после чего преобразуем её в тип `int` (`lround` преобразует в `long`).
`std::clamp` – функция, задающая диапазон значений и приводящая к нему все, что выходит за его пределы.
После остается только преобразовать полученные координаты в байты для передачи в правильно порядке. Делается это при помощи функции `htons (Host TO Network Short)`. Это функция, которая переводит 16-битное число (`uint16_t`) из формата компьютера в сетевой порядок байтов.

```
pack.x = htons((uint16_t)xi);
pack.y = htons((uint16_t)yi);
```

И отправляем байты при помощи созданной ранее функции на ESP:

```
sendData(&pack, sizeof(pack));
&pack – адрес, где лежат наши подготовленные байты.
sizeof(pack) – количество байтов.
```

Прием данных на ESP32 и вывод координат в консоль

Микроконтроллер ESP32 можно программировать в разных IDE, но для удобства будем использовать Arduino IDE. Загрузка будет при этом медленнее, чем в родной среде, зато привычный интерфейс.

Настройка ESP32

Для начала нужно установить плату ESP32 в список поддерживаемых. Для этого открываем Менеджер плат, вводим «ESP32» и ставим esp32 от Espressif Systems (**не от Arduino**). После этого открываем список плат, нажимаем «Выберите другую плату или порт» и вводим «esp32 dev module». Там много вариантов, отличающихся на 1 букву, выбираем именно **esp32 dev module**!
После этого на плату можно загружать код. Попробуйте запустить простейший скетч с миганием светодиода на 2 порту. Если код загружается,

хорошо. Если нет (это бывает на компьютерах с процессором AMD), то нужно при каждой загрузке вручную перезагружать плату в нужный момент:

1. Ждете, пока код сгенерируется и начнет загружаться;
2. Нажимаете на плате на кнопку BOOT;
3. После этого нажимаете на плате на кнопку RST;
4. Отпускаете кнопку RST;
5. Отпускаете кнопку BOOT.

Загрузка должна начаться.

Код для приёма данных

Составим код для приёма данных. Нам понадобятся 2 библиотеки:

```
#include <WiFi.h>      // Библиотека для работы с Wi-Fi
#include <WiFiUdp.h>   // Библиотека для работы с UDP-протоколом поверх Wi-Fi
```

Так как сеть будет создаваться именно на ESP, то нужно записать в переменные её имя и пароль.

```
const char* ssid = "ESP32_AP";
const char* password = "12345678"; // Пароль (минимум 8 символов)
```

Создаем объект класса и порт подключения (именно тот, что был указан в программе на C++).

```
WiFiUDP udp;
const unsigned int localPort = 1919;
```

И, наконец, создаем массив для приема байтов.

```
uint8_t buffer[4];
```

Начальные настройки `setup()`
Включаем Serial монитор

```
Serial.begin(115200);
```

Создаем сеть WiFi.

```
WiFi.softAP(ssid, password);
```

Выводим IP адрес в монитор

```
Serial.print("IP ESP32: ");
Serial.println(WiFi.softAPIP());
```

И открываем UDP-порт

```
udp.begin(localPort);
```

После этого в бесконечном цикле `loop()` обрабатываем данные.

Проверяем, пришёл ли UDP-пакет, если да, пишем его размер

```
int packetSize = udp.parsePacket();
```

Проверяем, что размер пакета ровно 4 байта. Если да, то обрабатываем:

```
if (packetSize == 4)
```

Читаем 4 байта из входящего пакета в массив `buffer`

```
udp.read(buffer, 4);
```

Восстанавливаем координату X из двух байтов

`buffer[0]` — старший байт
`buffer[1]` — младший байт

Сдвиг << 8 возвращает старший байт на своё место

```
uint16_t x = (buffer[0] << 8) | buffer[1];
```

Аналогично собираем координату Y

```
uint16_t y = (buffer[2] << 8) | buffer[3];
```

Выводим полученные координаты в Serial Monitor

```
Serial.print("X: ");
Serial.print(x);
Serial.print("  Y: ");
Serial.println(y);
```

Старшие и младшие байты

Многие числа занимают больше одного байта. Например, `uint16_t` (16-битное число) занимает **2 байта**.

Эти два байта называются:

- **старший байт (high byte)** — содержит более значимую часть числа
- **младший байт (low byte)** — содержит менее значимую часть числа

Например, число `0x1234` состоит из двух байтов:

`0x12` — старший байт,

`0x34` — младший байт.

При передаче по сети обычно используется порядок: сначала старший байт, затем младший.

Чтобы собрать число обратно из двух байтов, используется операция сдвига и объединения:

```
uint16_t value = (highByte << 8) | lowByte;
```

Сдвиг << 8 возвращает старший байт в его позицию, а оператор | объединяет оба байта в одно число.