МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки, специальности)

Компьютерные технологии и математическое моделирование
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему «Разработка алгоритмов движения группы беспилотных объектов»

Студент Н.Н. Герасимов
(И.О. Фамилия) (личная подпись)

Руководитель канд. техн. наук, доцент, Т.Г. Султанов
(ученая степень, звание, И.О. Фамилия)

Консультант М. В. Дайнеко
(ученая степень, звание, И.О. Фамилия)

Тольятти 2021

Аннотация

Выпускная квалификационная работа посвящена следующей теме: «Разработка алгоритма движения беспилотного объекта». Актуальность данной темы обусловлена повсеместным внедрением элементов автоматизации и роботизации в различные виды производства. Областью применения разработанного алгоритма могут быть различные складские помещения, а также цеха различных производств. Объектом исследования является задача классификации при помощи компьютерного зрения.

Целью данной работы является разработка и программная реализация алгоритма движения беспилотного объекта по линии фиксированной ширины.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Проанализировать способы распознавания различных объектов на изображении;
2. Выделить требования, предъявляемые к беспилотному объекту и его программному обеспечению;
3. Разработать алгоритм движения;
4. Выполнить программную реализацию алгоритма;
5. Провести тестирование разработанного алгоритма;
6. Сформировать вывод о полученных результатах.

В первом разделе выпускной квалификационной работы был произведён анализ уже существующих решений, затем на основе полученных данных были составлены требования, предъявляемые к беспилотному объекту и программному обеспечению, а также был разработан алгоритм движения. Второй раздел посвящена описанию выбранных инструментов программной реализации алгоритма и его непосредственной реализации. В третьем разделе описывается тестирование разработанной программы. Объём бакалаврской работы: 46 страниц, 41 рисунок, 3 формулы, 20 источников и 2 приложения.

Abstract

This graduation work is devoted to developing an unmanned flying vehicles movement algorithm. The relevance of this topic is explained by the widespread implementation of automation and robotic application elements in various types of production. The suggested algorithm can be used in different warehouses, storage facilities, as well as workshops of various industries.

The graduation work consists of an introduction, 3 chapters and a conclusion, as well as the list of 20 references including 9 foreign sources and 2 appendices. The object of the graduation work is the problem of classification using computer vision.

The aim of the research is to develop and implement a software algorithm of an unmanned flying vehicle movement along a fixed-width line.

In order to achieve the above-mentioned aim, the following objectives have to be attained:

1. to analyze the ways of recognizing various flying vehicles in an image;
2. to identify the requirements for an unmanned flying vehicle and the corresponding software;
3. to develop an unmanned flying vehicle movement algorithm;
4. to provide the implementation of the software algorithm;
5. to test the proposed software;
6. to draw up a conclusion about the results obtained.

The first chapter of the present graduation work develops an algorithm of the unmanned flying vehicle movement based on the use of a colour filter and the Boundary Following algorithm. This chapter also dwells on the requirements for an unmanned flying vehicle.

The second chapter of the research describes the selected tools for the software implementation of the algorithm.

The third chapter of the investigation deals with testing of the suggested software.

Содержание

Введение

Технический прогресс никогда не стоит на месте. И робототехника также стремительно развивается. Роботы уже занимают значительную часть в жизни общества. Их применяют во многих сферах: начиная от бытовых дел (робот-пылесос) и заканчивая различными производствами, где основную работу выполняет сам робот, а человек лишь контролирует процесс исполнения.

В России уже продолжительный период времени наблюдается тенденция полной автоматизации технологического процесса, однако вопрос о том, когда она завершится, будет открыт ещё много лет. Производства стараются частично или полностью автоматизировать те процессы, которые не требуют долгосрочных затрат.

Одним из примеров таких процессов является логистика внутри складских помещений. Производства оборудуют склады специальными автоматизированными погрузчиками и тележками, которые позволят заменить человека в цепочки процесса логистики.

Вопрос обеспечения возможности навигации внутри помещения является индивидуальным для каждого автоматизированного устройства. В одних случаях требуется специальным образом подготовить помещение при помощи разметки на полу, а в других требуется установить специальные метки на стенах и полках.

Актуальность бакалаврской работы обусловлена поиском такого программного обеспечения, которое позволит сократить затраты на внедрение автоматизированных и беспилотных объектов.

Цель данной работы – разработка алгоритма, который позволяет перемещаться беспилотному объекту по разметке заранее определённого маршрута.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Проанализировать способы нахождения контуров различных объектов на изображении;
2. Выделить требования, предъявляемые к беспилотному объекту и его программному обеспечению;
3. Разработать алгоритм движения;
4. Выполнить программную реализацию алгоритма;
5. Провести тестирование разработанного алгоритма;
6. Сформировать вывод о полученных результатах.

В первом разделе произведён анализ уже существующих алгоритмов движения беспилотных объектов на примере роботов, специализирующихся на перемещении грузов в складских помещениях. На основе полученных данных выработаны требования, предъявляемые как к беспилотному объекту, так и к программному обеспечению. На основе этих требований был разработан алгоритм движения беспилотного объекта.

Во втором разделе описана программная реализация двух программ, которые необходимы для работы и демонстрации разработанного алгоритма. Первая программа осуществляет калибровку цветового фильтра, значения, полученные в этой программе, необходимы для корректной работы второй программы. Вторая программа реализует разработанный алгоритм, а также его визуализирует при помощи графического интерфейса.

В третьем разделе выполнено тестирование разработанного программного обеспечения и проанализировано его быстродействие.

Разработанный алгоритм позволит упростить процесс автоматизации на предприятии, так как алгоритму требуется только наличие камеры, которую зафиксировали на определённой высоте и линия маршрута, по которой должен двигаться беспилотный объект. Наличие различных дорогостоящих датчиков (например, ультразвуковые датчики для определения расстояния) не требуются.

1 Разработка алгоритма движения беспилотного объекта

1.1 Проектирование алгоритма движения беспилотного объекта

Под беспилотным объектом подразумевается такой объект, который перемещается по поверхности при помощи колёс. Подразумевается, что такой объект двигается плавно по маршруту из предварительно нанесённой на полу линии фиксированной ширины. Маршрут допускает наличие поворотов на 90 градусов.

В каждый момент времени считывается изображение с камеры. Полученное изображение предварительно обрабатывается путём конвертации из RGB цветовой модели в HSV. Затем на изображение накладывается цветовой фильтр, который превращает полученный кадр в чёрно-белое изображение, такое изображение ещё называют бинарным из-за наличия всего двух цветов.

Преобразование кадра в бинарное изображение необходимо для Boundary Following алгоритма [1]. Этот алгоритм специализируется на поиске границ объекта в изображении. Данный алгоритм позволяет обнаружить искомую линию маршрута в кадре.

Особенность разработанного алгоритма движения беспилотного объекта заключается в том, что он проверяет необходимость смещения по горизонтали в том случае, если беспилотный объект ушёл с линии маршрута по причине внешних факторов. Для этого контур линии маршрута упрощается: берутся угловые точки и по ним строится прямоугольник. Прямоугольник необходим для того, чтобы вычислить координаты его центра, на основе которых вычисляется поправка на смещение.

Маршрут в кадре может быть трёх видов:

1. Прямая линия,
2. Прямая линия с поворотом налево под 90 градусов,
3. Прямая линия с поворотом направо под 90 градусов.

Если в кадре представлен первый случай, то вычисление поправки производится по следующей формуле:

$$delta = \frac{MAX\_WIDTH}{2} - x\_center, \quad (1)$$

где $delta$ - поправка на смещение по горизонтали;

MAX_WIDTH – ширина кадра в пикселях;

x_center – координата центра прямоугольника по оси абсцисс.

Если же в кадре обнаружен поворот, то необходимо для начала вычислить координату центра не целого прямоугольника, а только прямой линии, которая идёт непосредственно перед поворотом. Тогда будет применяться формула 2 для вычисления поправки при наличии поворота налево, а формула 3 – при наличии поворота направо.

$$delta = \frac{MAX\_WIDTH}{2} - (width - \frac{LINE\_WIDTH}{2} + x), \quad (2)$$

$$delta = \frac{MAX\_WIDTH}{2} - (x + \frac{LINE\_WIDTH}{2}), \quad (3)$$

где $LINE\_WIDTH$ – ширина линии маршрута в пикселях;

x – координата левого верхнего угла прямоугольника по оси абсцисс.

Полученная поправка может быть как положительной, так и отрицательной. Если она положительная, то это означает, что надо сместиться влево по горизонтали, в противном случае надо сместиться вправо.

Поворот определяется следующим образом. Если ширина прямоугольника больше, чем ширина линии, и в тоже время высота прямоугольника меньше или равна ширине линии, значит беспилотный объект достиг места, где необходим поворот. В этом случае проверяется местонахождение центра прямоугольника по оси абсцисс. Если центр

находится левее центра кадра, то осуществляется поворот налево, в противном случае – поворот направо.

На рисунке 1 изображена UML диаграмма разработанного алгоритма движения беспилотного объекта.

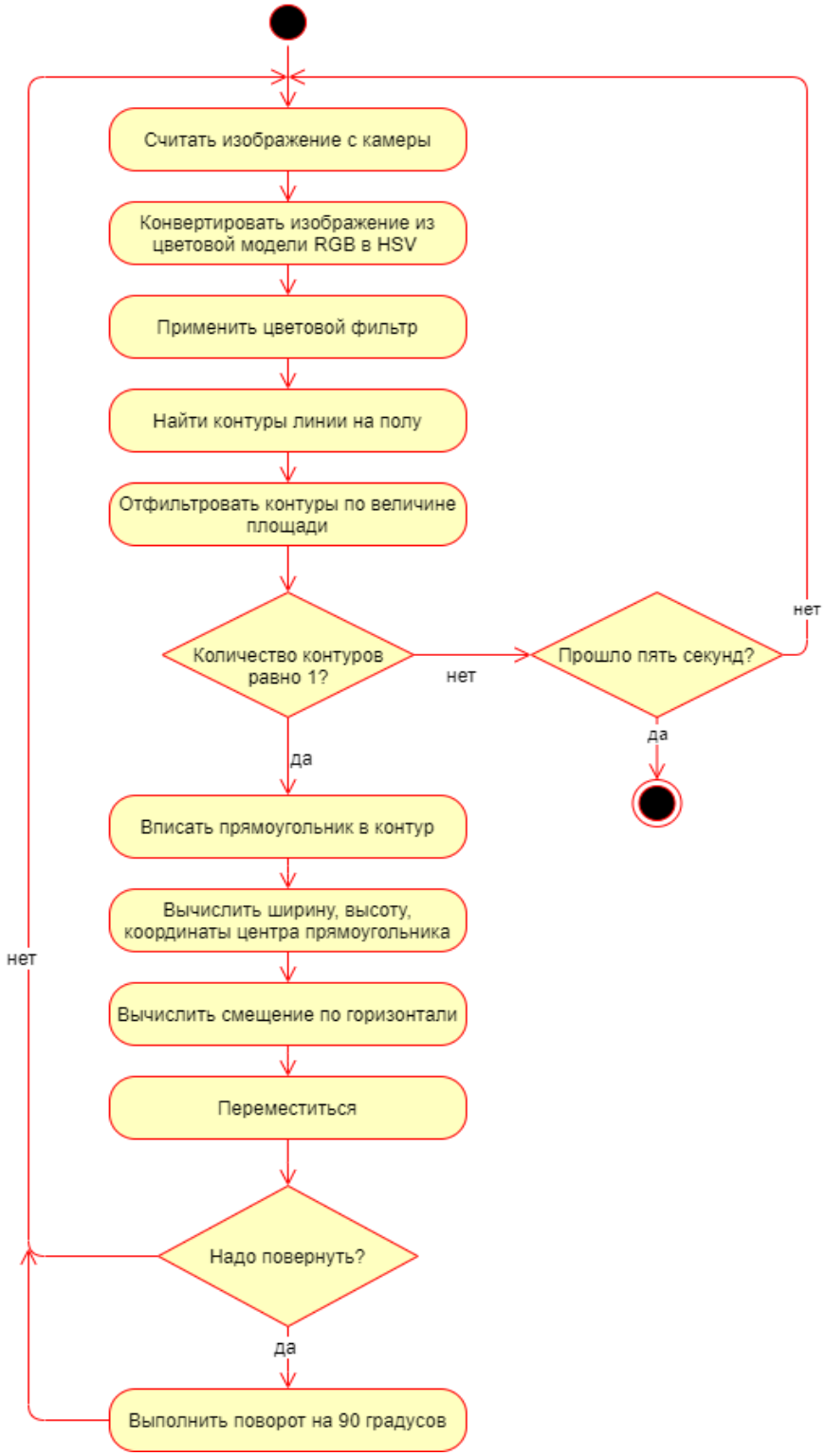

Рисунок 1 – UML диаграмма алгоритма движения беспилотного объекта

Теперь разберём такие основные компоненты алгоритма как цветовой фильтр и поиск контура поподробнее.

1.2 Цветовой фильтр

Чтобы найти на изображении линию маршрута необходимого цвета, надо для начала найти все прямоугольные объекты, затем проверить их цвета. Но потом возникает вопрос: как найти край искомой линии? Поиск такого места на каждом кадре видеопотока достаточно сложная вычислительная задача, поэтому необходимо упростить задачу [3].

Для этого необходимо воспользоваться цветовым фильтром, который будет использовать цветовую модель HSV. Аббревиатура HSV образуется из слов hue (тон), saturation (насыщенность) и value (значение). Почему необходимо использовать именно эту цветовую модель вместо стандартной RGB? Особенность RGB модели состоит в том, что цвета получаются путём смешивания красного, зелёного и синего цветов, однако она является аппаратно-зависимой [15]. То есть для однозначного определения цвета на изображении необходимо также учитывать характеристики камеры. В тоже время цветовая модель HSV лишена данного недостатка. Также цветовая модель HSV отображает информацию о цвете в более привычной человеку форме: Что это за цвет? Насколько он насыщенный? Насколько он светлый или тёмный?

Перевод из RGB в HSV происходит следующим образом.

Считаем, что:

$$H \in [0, 360]$$
$$S, V, R, G, B \in [0,1]$$

Пусть MAX – максимальное значение из R, G, B, а MIN – минимальное из них. Тогда значение тона (hue) будет определяться формулой 4.

$$H = \begin{cases} \text{не определено, если } MAX = MIN \\ 60 * \dfrac{G - B}{MAX - MIN}, \text{если } MAX = R \text{ и } G \geq B \\ 60 * \dfrac{G - B}{MAX - MIN} + 360, \text{если } MAX = R \text{ и } G < B \\ 60 * \dfrac{B - R}{MAX - MIN} + 120, \text{если } MAX = G \\ 60 * \dfrac{R - G}{MAX - MIN} + 240, \text{если } MAX = B \end{cases} \quad (4)$$

Значение насыщенности (saturation) задаётся формулой 5:

$$S = \begin{cases} 0, \text{если } MAX = 0 \\ 1 - \dfrac{MIN}{MAX}, \text{в остальных случаях} \end{cases} \quad (5)$$

А значение цвета (value) задаётся формулой 6.

$$V = MAX \quad (6)$$

После того, как изображение было преобразовано в цветовую модель HSV, начинается непосредственна фильтрация. Цветовой фильтр имеет два граничных значения HSV_{min} и HSV_{max}. Если хотя бы одна составляющая цвета у пикселя меньше, чем аналогичные граничные значения HSV_{min} или же больше, чем значения HSV_{max}, то такой пиксель считается за границей искомого цвета, и он заменяется чёрным цветом. Если пиксель имеет цвет в диапазоне фильтра, то он заменяется белым цветом.

Таким образом, после применения цветового фильтра само изображение станет чёрно-белым, при этом белым цветом будет обозначаться искомая линия маршрута, а чёрным цветом – весь остальной фон. При таком подходе получается добиться обнаружение линии маршрута в 100% случаях.

1.3 Алгоритм поиска контуров на изображении

Ядром разработанного алгоритма служит поиск контуров линии маршрута в кадре. Это можно добиться при помощи алгоритма под названием Boundary Following (Граничное отслеживание).

Для алгоритма требуется наличие бинарного изображения. Бинарное изображение – это такое изображение, где пиксели представлены в виде 0 и 1. 0 ставится тому пикселю, который относится к пустому пространству или фону, а 1 ставится в соответствии с пикселем, который относится к объекту. Пример бинарного изображения показан на рисунке 2. Здесь серым цветом показан некоторый объект. В дальнейших рисунках нули убраны, так как они не имеют смысловой нагрузки.

0	0	0	0	0	0	0
0	0	1	1	1	1	0
0	1	0	0	1	0	0
0	0	1	0	1	0	0
0	1	0	0	1	0	0
0	1	1	1	1	0	0
0	0	0	0	0	0	0

Рисунок 2 – Пример бинарного изображения

Введём необходимые обозначения. Точки b_i обозначают контур (пиксели со значением 1), а точки c_i обозначают фон (пиксели со значением 0). Каждая точка b_i имеет 8 смежных точек, которые обозначаются $n_1 \ldots n_8$. Нумерация смежных с b_i точек начинается от точки c_i и продолжается по направлению часовой стрелки.

Теперь перейдём непосредственно к алгоритму.

Шаг 1. Найти самую верхнюю и крайнюю левую точку, значение которой равно 1. Обозначить её как b_0. Смежную слева с этой точкой обозначить как c_0. Если это невозможно сделать из-за того, что b_0 находится у границы изображения, то найти ближайшую точку, начиная слева и продолжая в направлении движения часовой стрелки. На рисунке 3 показан первый шаг алгоритма.

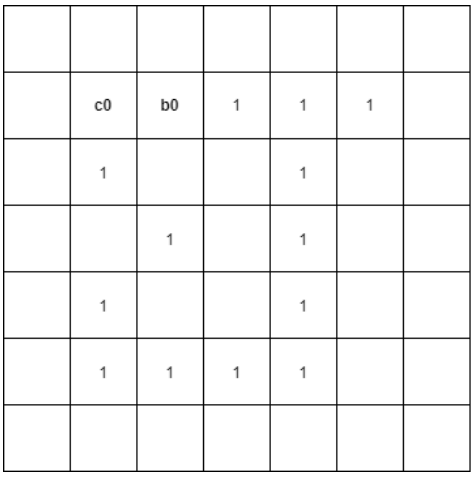

Рисунок 3 – Первый шаг алгоритма поиска контуров

Шаг 2. Найти первую точку n_i, которая имеет значение 1. Поиск начинается с точки c_0 и также продолжается в направлении движения часовой стрелки (рисунок 4).

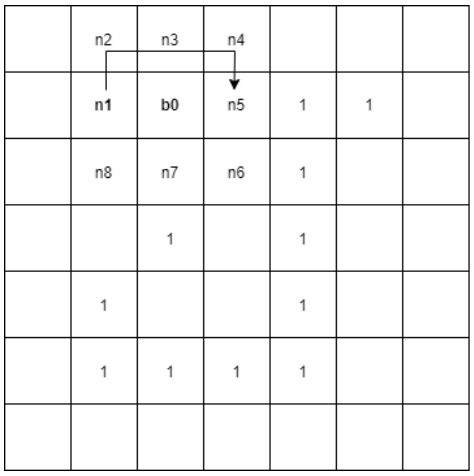

Рисунок 4 – Поиск ближайшей точки контура

Шаг 3. Найденную точку n_i обозначить как b_1, а точку n_{i-1} обозначить c_1 (рисунок 5). Теперь найденные точки b_0 и b_1 необходимо запомнить.

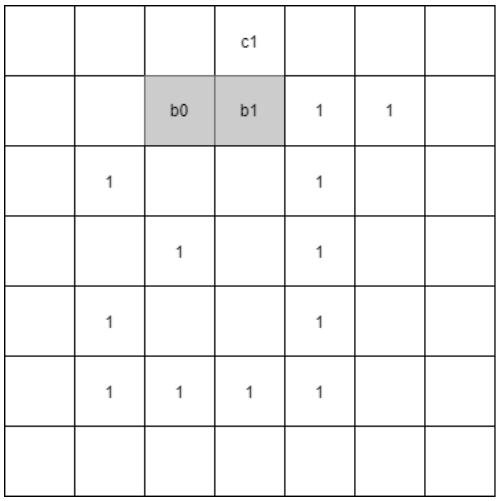

Рисунок 5 – Запоминание найденной точки контура

Шаг 4. Повторять шаги 2 и 3 до тех пор, пока точка b_i не равна точке b_0 и следующая граничная точка не будет равна b_1.

В результате проделанных шагов получится упорядоченная последовательность точек b. Эта последовательность и является найденным контуром (рисунок 6).

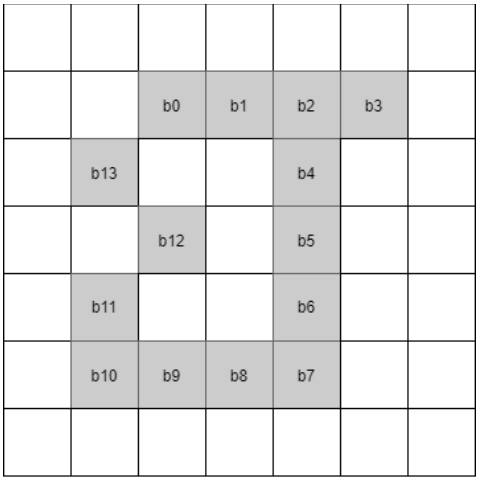

Рисунок 6 – Результат работы алгоритма

Алгоритм Boundary Following позволяет достаточно точно определять контуры различных объектов в бинарных изображениях.

1.4 Требования, предъявляемые к беспилотному объекту

Разработанный алгоритм движения беспилотного объекта накладывает следующие требования на беспилотный объект:

- наличие камеры с разрешением матрицы не менее 640x480 пикселей;
- частота съёмки камеры не менее 30 кадров в секунду;
- камера должна быть зафиксирована на определённой высоте от пола;
- расстояние от камеры до пола не должно изменяться в процессе движения.

Выполнение вышеперечисленных требований гарантирует правильную работоспособность алгоритма движения беспилотного объекта.

Выводы и результаты по первому разделу

В данном разделе был разработан алгоритм движения беспилотного объекта. Алгоритм был описан при помощи UML диаграммы деятельности (activity diagram) и дополнен текстовым описанием. Также были описаны принципы работы цветового фильтра и алгоритма поиска контуров линии маршрута.

Из рисунка 1 можно видеть, что основная работа алгоритма будет заключаться в повторении цикла до тех пор, пока в течение пяти секунд не будет обнаружено хотя бы одного контура.

Самым важным моментом является то, что для успешной работы алгоритма от беспилотного объекта требуется всего лишь наличие фиксированной камеры. При этом нет нужды в различных ультразвуковых датчиках, что существенно снижает стоимость такого аппарата.

2 Программная реализация алгоритма движения беспилотного объекта

2.1 Описание используемых инструментов

Было принято решение написать алгоритм на языке программирования Python, потому что данный язык имеет очень мощные инструменты для работы с обработкой изображения. Также язык позволяет писать компактные программы, не уступая при этом в их функциональности. Для программной реализации необходимы три библиотеки: NumPy, OpenCV, PyGame.

Библиотека NumPy является библиотекой с открытым исходным кодом. NumPy – это сокращение от Numeric Python. Название даёт понять, что эта библиотека предоставляет функционал, которые реализует различные математические операции. Основным объектом библиотеки является тип array. Данный тип похож на стандартный список в языке Python, только за одним исключением: в нём обязательно должны быть элементы одного типа (в объекте типа list можно хранить элементы разных типов). Список в Python имеет существенный недостаток: низкая скорость вычислительных операций с большим объёмом данным. Библиотека NumPy устраняет этот недостаток, выполняя числовые операции в разы быстрее и, главное, намного эффективнее и удобнее [5].

Далее идёт open source библиотека компьютерного зрения OpenCV. Её основной целью служит анализ, классификация и обработка изображения [7][8]. В неё входят более 2500 алгоритмов, среди которых алгоритмы для интерпретации изображений, калибровки камеры по эталону, устранение оптических искажений, определение формы объекта и другие. Она имеет реализацию на различных языках: Python, C++, Java, Matlab. OpenCV состоит из пяти модулей:

– cxcore содержит базовые операции над многомерными числовыми массивами, матричную алгебру и базовые функции 2D графики;

– CV является модулем для обработки изображений и компьютерного зрения;

– HighGUI предоставляет инструмент для создания пользовательского интерфейса;

– Cvaux содержит экспериментальные и устаревшие функции;

– CvCam позволяет осуществлять захват видеоизображений.

В данной выпускной квалификационной работе будут использоваться только три модуля: CV, HighGUI и CvCam.

Так как в рамках данной выпускной квалификационной работы разрабатывается только алгоритм движения, то его необходимо визуализировать. Для этих целей применяется библиотека PyGame [9]. Она позволит выполнить компьютерное моделирование движущегося беспилотного объекта.

2.2 Калибровка цветового фильтра

Особенность работы алгоритма заключается в том, что нет привязанности к определённому цвету линии, по которому будет двигаться беспилотный объект. Данная особенность обусловлена наличием цветового фильтра. Для чего здесь применяется цветовой фильтр? Как было описано раннее, задача нахождения контура является сложным вычислительным процессом.

Беспилотник двигается постоянно и равномерно, поэтому необходимо найти такое решение, которое позволит максимально быстро обрабатывать изображение. Для этих целей применяется цветовой фильтр.

Принцип его работы заключается в следующем: сначала считывается изображение с камеры, затем оно конвертируется в HSV цветовую модель, после этого накладывается фильтр на изображение, т.е. все цвета, которые не входят в диапазон указанных значений, на изображении будут иметь чёрный цвет, а остальные будут показаны белым цветом [4]. В результате

разноцветное изображение станет чёрно-белым. В таком виде поиск линии маршрута происходит значительно быстрее, особенно при небольшом разрешении изображения (например, от 640x480 до 1280x720 пикселей).

Для калибровки цветового фильтра была написана специальная программа, листинг которой представлен в Приложении А.

При запуске программы для калибровки появляется два окна: центральное окно с названием «result», куда выводится изображение с камеры и маленькое окно «settings» с 6 ползунками для управления значениями нижней и верхней границ цвета. Задача этой программы заключается в том, чтобы выставить положения ползунков таким образом, чтобы искомая линия маршрута была чисто белого цвета.

Правильно настроенный цветовой фильтр позволяет избавиться от лишних шумов на изображении, что в конечном счёте повышает точность обнаружения линии маршрута до 100%.

Разработанная программа позволяет настроить цветовой фильтр таким образом, чтобы можно было однозначно определять контуры объекта искомого цвета. При этом наличие посторонних предметов других цветов никак не влияет на качество поиска объекта искомого цвета.

2.3 Принцип работы основной программы

В первую очередь инициализируются переменные со значениями ширины и высоты изображения в пикселях, также задаётся ширина линии маршрута в пикселях [6]. Затем инициализируется камера и выставляется её разрешение. Листинг этих операций представлен на рисунке 7.

После этого подключается главное окно PyGame для отображения пройденной траектории. Переменная surface содержит себе поверхность, на которой будут отрисовываться различные объекты. Также инициализируется переменная clock, которая позволит управлять частотой отрисовки кадров (рисунок 8).

```
# инициализация параметров изображения
MAX_WIDTH = np.int32(1280)
MAX_HEIGHT = np.int32(720)
LINE_WIDTH = 160

# инициализация камеры
cv2.namedWindow("result")
webcam = cv2.VideoCapture(0)
webcam.set(3, MAX_WIDTH)
webcam.set(4, MAX_HEIGHT)
```

Рисунок 7 – Листинг инициализации параметров изображения и подключения камеры

```
# инициализация pygame
pygame.init()
surface = pygame.display.set_mode([MAX_WIDTH, MAX_HEIGHT])
clock = pygame.time.Clock()
```

Рисунок 8 – Инициализация элементов библиотеки PyGame

Далее идёт настройка цветового фильтра (рисунок 9). Как раз здесь необходимо использовать те значения, которые получили при калибровке цветового фильтра из пункта 2.2. В переменную hsv_min записываются значения первых трёх ползунков калибровочной программы, то есть там будет массив [h1, s1, v1]. Аналогичным образом записывается переменная hsv_max.

```
# определение цветового фильтра
hsv_min = np.array((0, 141, 0), np.uint8)
hsv_max = np.array((14, 255, 255), np.uint8)
```

Рисунок 9 – Настройка цветового фильтра

После этого выставляются значения для переменных, при помощи которых будет отображаться пройденная траектория (рисунок 10). Беспилотный объект обозначается зелёным квадратом, стороны которого равны SIZE пикселей. Переменные i и j обозначают координаты текущей позиции беспилотного объекта по оси абсцисс и ординат соответственно. Переменная step нужна для того, чтобы указать на сколько пикселей переместится квадрат в единичный момент времени. Переменная delta содержит поправку на смещение в горизонтальной плоскости, а height – величину расстояния, которое необходимо пройти вперёд.

```
# переменные для отображения траектории
i, j, SIZE = MAX_WIDTH/2, MAX_HEIGHT-50, 50
step, delta, height, angle = 1, 0, 0, 0
dots = np.array([[MAX_WIDTH/2, MAX_HEIGHT-50]])
dots_for_display = np.array([[MAX_WIDTH/2, MAX_HEIGHT-50]])
coords = {"x": 0, "y": 0}
dirs = {0:   [["y", -1], ["x",  1]],
        90:  [["x", -1], ["y", -1]],
        180: [["y",  1], ["x", -1]],
        270: [["x",  1], ["y",  1]]}
```

Рисунок 10 – Инициализация переменных для отображения траектории

Переменна angle хранит актуальное направление движения в градусах. По умолчанию значение равно 0, то есть движение идёт в северном направлении. Если угол увеличивается, значит поворот идёт против часовой стрелки. Система углов показана на рисунке 11.

Массив dots содержит координаты точек, на которых беспилотник совершил остановку. Причин остановки может быть несколько:

– на изображении не обнаружено ни одного контура, характеризующего наличие линии маршрута;
– $height \leq LINE\_WIDTH$;
– необходимо выполнить поворот.

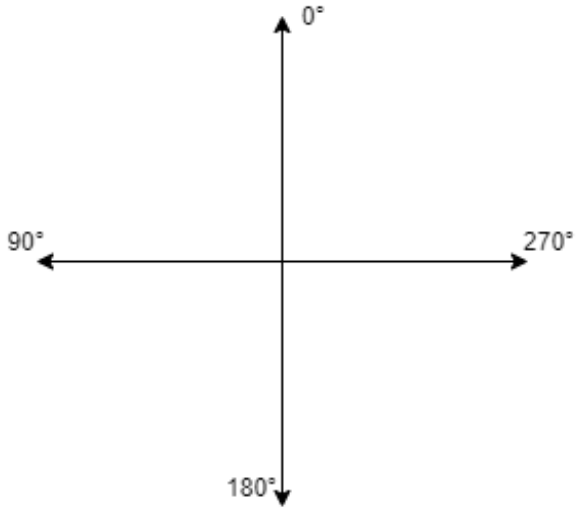

Рисунок 11 – Система углов, применяемая в моделировании траектории движения беспилотного объекта

Для массива dots_for_display используется такая логика: берутся все точки из массива dots, в конец добавляется точка текущей позиции, затем рисуется красная ломаная линия, которая соединяет все точки из массива dots_for_display. Объект coords содержит два поля: x и y, в этих полях указывается смещение координат по соответствующим осям относительно последней запомненной точки из массива dots. Объект dirs содержит необходимую информацию для корректного вычисления координат при различных углах поворота. Здесь значению угла поворота относительно севера ставится в соответствие две пары значений. Первая пара значений содержит информацию о том, по какой оси и в каком направлении необходимо передвигать зелёный квадрат при моделировании движения вперёд, вторая же пара значений применяется при движении вправо. Например, если текущий угол поворота равен нулю, тогда при движении вперёд необходимо уменьшать координату по оси ординат и при движении вправо надо увеличивать координату по оси абсцисс. Такое поведение обусловлено системой координат, которая применяется в библиотеке PyGame [12]. На рисунке 12 показаны координаты вершин прямоугольника, размер которого равен 120x80 пикселей.

Рисунок 12 – Демонстрация системы координат, применяемой в библиотеке PyGame

В данной программе есть возможность передвинуть нарисованную траекторию при помощи нажатия и удерживания любой кнопки мыши. Для выполнения этой функции используется логический флаг MOVING, также здесь задаётся логический флаг для начала отсчёта пяти секунд в случае отсутствия искомых контуров на изображении, а переменная start_ticks запоминает непосредственно момент начала отсчёта (рисунок 13). На этом подготовительный этап завершён.

```
MOVING, TIMEOUT_START = False, False
start_ticks = 0
```

Рисунок 13 – Инициализация логических флагов и переменной для хранения момента начала отсчёта пяти секунд

Затем начинается основной цикл программы (рисунок 14). В первую очередь вычисляются координаты текущей позиции, то есть к переменным i и j суммируются накопленные смещения по осям X и Y из объекта coords. Далее идёт проверка на момент остановки. Для этого сравнивается высота прямоугольника с шириной линии: если условие выполняется, значит беспилотный объект достиг конца траектории или ему необходимо повернуть.

```
while True:
    # вычисление координат текущей позиции
    currentPos = [i + coords["x"], j + coords["y"]]
    # проверка на момент остановки
    if height ⩽ LINE_WIDTH:
        coords["x"], coords["y"] = 0, 0
        if currentPos not in dots:
            dots = np.vstack((dots, currentPos))
            i, j = currentPos
    elif height > LINE_WIDTH:
        coords[dirs[angle][0][0]] += step * dirs[angle][0][1]
```

Рисунок 14 – Проверка на момент остановки

В таком случае необходимо запомнить текущую позицию, то есть записать её в массив dots, при этом исключая наличие дубликатов. Если height всё-таки больше ширины линии, тогда в объект coords надо прибавить смещение на один шаг в той оси координат, которая соответствует направлению движения вперёд относительно текущего угла поворота. В итоге, перемещение вперёд будет осуществляться по шагам до тех пор, пока значение height будет больше ширины линии. Пошаговое перемещение используется по той причине, что, во-первых, беспилотный объект не знает длину текущей секции маршрута, а, во-вторых, это позволит оперативно реагировать на изменение ситуации в кадре: например, коробка упала на линию маршрута, тогда беспилотник остановится на расстоянии, равном ширине линии, и не будет пытаться пройти сквозь препятствие.

Затем вычисляется поправка на смещение по горизонтали, чтобы беспилотный объект стремился к центру линии (рисунок 15). Беспилотник начинает смещение только в том случае, если абсолютная величина поправки больше 20. Это ограничение сделано для того, чтобы компенсировать погрешности в тех механизмах, которые будут задействованы для непосредственного движения. Если delta имеет отрицательное значение,

значит линия маршрута находится справа, соответственно беспилотнику нужно сместиться вправо.

```
# смещение по горизонтали
if abs(delta) > 20:
    diff = step * dirs[angle][1][1] if delta < 0 else -step * dirs[angle][1][1]
    coords[dirs[angle][1][0]] += diff
```

Рисунок 15 – Вычисление поправки на смещение по горизонтали

На рисунке 16 показан принцип заполнения массива dots_for_display. Сначала в него копируются все значения из массива dots, а затем в конец дописывается текущая позиция. Необходимость хранить точки в двух массивах одновременно обусловлено тем, что в каждый момент времени изменяется только последняя секция маршрута. Под секцией подразумевается участок пути от одной точки к другой. Если же всё хранить в одном массиве, то точки остановок и поворотов будут перемешаны с текущей позицией, что в конечном итоге исказит отображаемый маршрут.

```
dots_for_display = dots.copy()
dots_for_display = np.vstack((dots_for_display, currentPos))
```

Рисунок 16 – Заполнение массива dots_for_display

Далее идёт непосредственно отрисовка самого маршрута при помощи библиотеки PyGame (рисунок 17). При помощи функции fill у переменной surface фон окна закрашивается в серый цвет. Затем применяется функция pygame.draw.aalines для отрисовки траектории пройденного маршрута. Она принимает 4 аргумента:

1. Поверхность, на которой надо нарисовать линии;
2. Цвет линий;

3. Булево значение, если передано значение True, то последняя точка будет соединена с первой прямой линией;

4. Массив с координатами всех точек.

Затем при помощи функции pygame.draw.rect отрисовывается текущая позиция беспилотного объекта. Для неё необходимо также передать поверхность для отрисовки, цвет и массив из 4 значений: координаты X и Y центра, длину и ширину.

```
# отрисовка пройденного маршрута
surface.fill(pygame.Color("gray"))
pygame.draw.aalines(surface, pygame.Color("Red"), False, dots_for_display)
pygame.draw.rect(surface, pygame.Color("Green"), currentPos + [SIZE, SIZE])
```

Рисунок 17 – Отрисовка пройденного маршрута

Далее считывается изображение с камеры, конвертируется полученное изображение в HSV цветовую модель, а при помощи функции cv2.inRange на изображение применяется цветовой фильтр (рисунок 18) [18]. В результате выполнения данного фрагмента в переменную thresh помещается бинарное изображение, то есть изображение, состоящее только из белых и чёрных пикселей.

```
# считывание изображение с камеры и применение цветового фильтра
flag, img = webcam.read()
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
thresh = cv2.inRange(hsv, hsv_min, hsv_max)
```

Рисунок 18 – Получение бинарного изображения

Затем при помощи функции cv2.findContours ищутся контуры линии маршрута (рисунок 19). Данная функция реализует описанный в пункте 1.3 Boundary Following алгоритм. Эта функции принимает в качестве первого

параметра бинарное изображение, полученное выше. Второй параметр позволяет выбрать режим алгоритма поиска контура [10]. RETR_TREE обозначает такой режим, которым извлекает все контуры и восстанавливает полную иерархию вложенных контуров. То есть самый первый контур будет внешний, а все последующие будут уже внутри первого контура. Третий параметр выбирает алгоритм аппроксимации контура [11]. CHAIN_APPROX_SIMPLE упрощает линию контура, беря в расчёт только начальную и конечную точки. После того, как контуры будут найдены, их необходимо отфильтровать по площади, чтобы избежать реагирования на лишние объекты, которые попадают в кадр.

```
# поиск и фильтрация контуров
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
filtered = [c for c in contours if cv2.contourArea(c) > 5000]
```

Рисунок 19 – Поиск и фильтрация контуров

Дальше идёт два варианта развития событий в зависимости от количества найденных контуров. Рассмотрим ситуацию, когда был обнаружен всего один контур (рисунок 20 и 21). В таком случае сначала сбрасывается логический флаг для начала отсчёта, потом в найденный контур вписывается прямоугольник при помощи функции cv2.boundingRect. Эта функция возвращает 4 значения: координаты X и Y верхнего левого угла, ширину и высоту вписанного прямоугольника. Затем на основе полученных данных вычисляется поправка на смещение по горизонтали по формулам 1-3.

Затем выполняется проверка на необходимость сделать поворот (рисунок 21). В том случае, если поворот осуществляется, то также, как и на рисунке 14, выполняется запоминание последней точки, только в данном фрагменте кода ещё изменяется значение angle на +90 или –90.

```
if len(filtered) == 1:
    TIMEOUT_START = False
    # вычисление координат левого верхнего угла, ширины и высоты прямоугольника
    x, y, width, height = cv2.boundingRect(filtered[0])
    # вычисление координат центра прямоугольника
    x_center, y_center = int(x + width / 2), int(y + height / 2)
    # вычисление поправки на смещение по горизонтали
    if width <= LINE_WIDTH:
        delta = int(MAX_WIDTH / 2 - x_center)
    elif width > LINE_WIDTH and x_center < MAX_WIDTH / 2:
        delta = int(MAX_WIDTH / 2 - (width - LINE_WIDTH/2 + x))
    elif width > LINE_WIDTH and x_center > MAX_WIDTH / 2:
        delta = int(MAX_WIDTH / 2 - (x + LINE_WIDTH/2))
    else:
        delta = 0
```

Рисунок 20 – Вычисление поправки на смещение

```
# проверка на поворот
if width > LINE_WIDTH >= height:
    coords["x"], coords["y"] = 0, 0
    if currentPos not in dots:
        dots = np.vstack((dots, currentPos))
        i, j = currentPos
    angle = (angle + 90) % 360 if x_center < MAX_WIDTH / 2 else (angle - 90) % 360
```

Рисунок 21 – Проверка на необходимость поворота

После выполненных действий в консоль выводится техническая информация, содержащая ширину и высоту найденного прямоугольника, величину delta и площадь контура (рисунок 22). Эти данные необходимы для того, чтобы откалибровать значения.

```
# вывод технической информации
print("w = %d h = %d d = %d a = %d" % (width, height, delta, int(cv2.contourArea(filtered[0]))))
```

Рисунок 22 – Вывод технической информации

На рисунке 23 показан фрагмент кода, который рисует найденный контур, центр прямоугольника и текст с его координатами в формате «x-y». В функцию cv2.rectangle сначала передаётся изображение, на котором необходимо нарисовать прямоугольник, потом координаты левого верхнего и правого нижнего углов. Затем передаётся цвет в формате BGR (Blue, Green, Red) со значениями в диапазоне от 0 до 255. В данном случае используется синий цвет. А последний параметр указывает толщину линии в пикселях.

```
# отрисовка контура и его центра
cv2.rectangle(img, (x, y), (x + width, y + height), (255, 0, 0), 2)
cv2.circle(img, (x_center, y_center), 10, (0, 0, 255), -1)
cv2.putText(img, "%d-%d" % (x_center, y_center), (x_center + 10, y_center - 10),
            cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 255), 2)
```

Рисунок 23 – Отрисовка графических элементов

Функция cv2.circle имеет такой же первый аргумент, как у предыдущей функции, затем указываются координаты центра, радиус окружности и её цвет. Последний аргумент также указывает толщину линии, если указать -1, то окружность будет полностью закрашена указанным цветом.

Функция cv2.putText используется для отрисовки текста на изображении. Туда необходимо передать изображение, выводимый текст, координаты самого текста на изображении, шрифт, величину масштаб (1 обозначает 100%), цвет и толщину букв в пикселях.

На этом закончились действия, которые выполняются в случае обнаружения одного контура. Рассмотрим противоположную ситуацию, когда не было найдено ни одного контура или было найдено от двух и более контуров. Фрагмент кода, обрабатывающий эту ситуацию представлен на рисунке 24. Сначала обнуляются переменные width, height, delta, чтобы остановить движение. Затем осуществляется проверка на отсутствие контуров в течение 5 секунд. Если до этого были обнаружен контур, тогда в переменную start_ticks записывается текущее количество тиков процессора. Что такое тик

процессора? Это единица измерения внутреннего системного времени [17]. Сколько миллисекунд составляет один тик процессора зависит от операционной системы. Затем проверяется разница между текущим количеством тиков и запомненным в start_ticks и делится на 1000, чтобы перевести в секунды. Если прошло 5 секунд, то цикл while прерывается.

```
else:
    # проверка на отсутствие контуров в течение пяти секунд
    width, height, delta = 0, 0, 0
    if not TIMEOUT_START:
        start_ticks = pygame.time.get_ticks()
        TIMEOUT_START = True
    else:
        seconds = (pygame.time.get_ticks() - start_ticks) / 1000
        if seconds > 5:
            print("Timeout")
            break
```

Рисунок 24 – Обработка случая, когда не был найден только один контур

Далее в окно с названием result выводится изображение img, на котором нарисованы контуры найденного прямоугольника, его центр и текст с координатами. Также необходимо обновить экран с отрисованным маршрутом при помощи функции pygame.display.flip и указать, сколько миллисекунд прошло с последнего вызова функции clock.tick. Указанное число 30 говорит о том, что в секунду будет отрисовано 30 кадров. Описанный фрагмент показан на рисунке 25.

```
cv2.imshow('result', img)

pygame.display.flip()
clock.tick(30)
```

Рисунок 25 – Вывод обработанного изображения и обновление экрана

Последние действия, которые выполняются в цикле, это проверка на нажатие клавиши Escape для досрочного выхода из программы [19]. Также здесь отслеживается нажатие на кнопку мыши и передвижения [20]. Этот функционал нужен для того, чтобы можно было посмотреть весь пройденный маршрут даже в том случае, если он выходит за границы отображаемого окна. Листинг этого фрагмента кода показан на рисунке 26.

```
key = pygame.key.get_pressed()
if key[pygame.K_ESCAPE]:
    break

for event in pygame.event.get():
    if event.type == pygame.QUIT:
        exit()

    if event.type == pygame.MOUSEBUTTONDOWN:
        MOVING = True
    if event.type == pygame.MOUSEBUTTONUP:
        MOVING = False

    if event.type == pygame.MOUSEMOTION and MOVING:
        dots = np.add(dots, event.rel)
        dots_for_display = np.add(dots_for_display, event.rel)
        i, j = np.add((i, j), event.rel)
```

Рисунок 26 – Обработка событий нажатии на кнопки

После выхода из цикла необходимо освободить камеру от захвата видеопотока и закрыть все окна (рисунок 27).

```
webcam.release()
cv2.destroyAllWindows()
```

Рисунок 27 – Освобождение камеры и закрытие всех окон

В приложении Б приводится полный код разработанной программы для моделирования алгоритма движения беспилотного объекта.

Выводы и результаты по второму разделу

Подведём следующие выводы:

– для выполнения программной реализации был выбран язык программирования Python и три библиотеки, которые предоставляют необходимый для выполнения задач функционал: NumPy, PyGame и OpenCV;

– было разработано две программы: первая программа необходима для калибровки цветового фильтра, при этом фильтр можно настроить под любой цвет, единственное требование заключается в том, что необходимо, чтобы объект контрастировал с остальным фоном; вторая программа использует данные откалиброванного цветового фильтра и реализует разработанные алгоритм и моделирует движение беспилотного объекта при помощи библиотеки PyGame;

– разработанные программы требуют минимальных вычислительных ресурсов, благодаря оптимизированному коду: в момент вычисления программы используют около 50 мегабайт оперативной памяти, а размер двух программ вместе занимает всего 7318 байт, что позволит запустить программное обеспечение на таких компактных устройствах, как Raspberry Pi 4 [14].

3 Тестирование разработанной программы

3.1 Тестирование разработанной программы для моделирования алгоритма движения беспилотного объекта

Для тестирования работоспособности разработанной программы необходимо наличие трёх вещей:

1. Устройство, на котором возможно запустить код на языке Python с предустановленными пакетами;
2. Камера;
3. Изображение маршрута в виде линии (наличие поворотов на 90 градусов опционально).

Программы запускали на стационарном компьютере. К компьютеру подключена веб-камера Canyon CNS-CWC6N [16]. Данная камера имеет максимальное разрешение кадра до 2048x1536 пикселей, но в данном тесте будет установлено разрешение 1280x720.

Далее был сымитирован маршрут при помощи нарисованной на листе бумаги формата А4 линии красным маркером (рисунок 28).

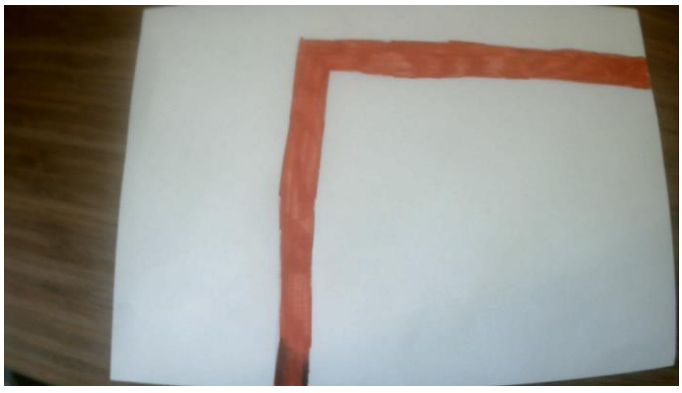

Рисунок 28 – Имитация линии маршрута

В первую очередь необходимо настроить цветовой фильтр при помощи калибровочной программы (Приложение А). В результате калибровки были

определены необходимые показатели для граничных условий HSV фильтра: hsv_min имеет hue = 0, saturation = 84, value = 116, а hsv_max имеет hue = 14, saturation = 255, value = 255. На рисунке 29 показан правильно настроенный цветовой фильтр. Полученные значения необходимо запомнить и использовать в основной программе (Приложение Б).

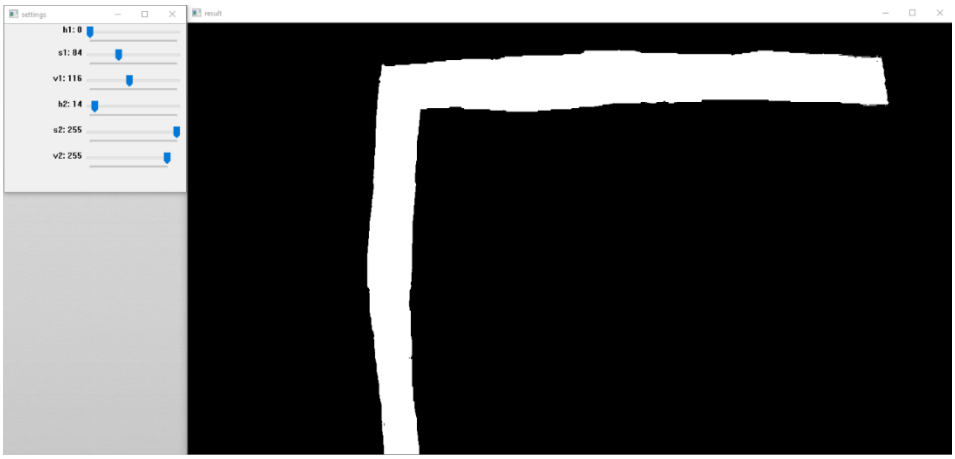

Рисунок 29 – Правильно настроенный цветовой фильтр

Теперь протестируем, определяет ли Boundary Following алгоритм саму линию маршрута, после конвертации кадра с камеры в бинарное изображение. На рисунке 30 видно, что программа отрисовала на изображении контур линии маршрута и определила координаты его центра.

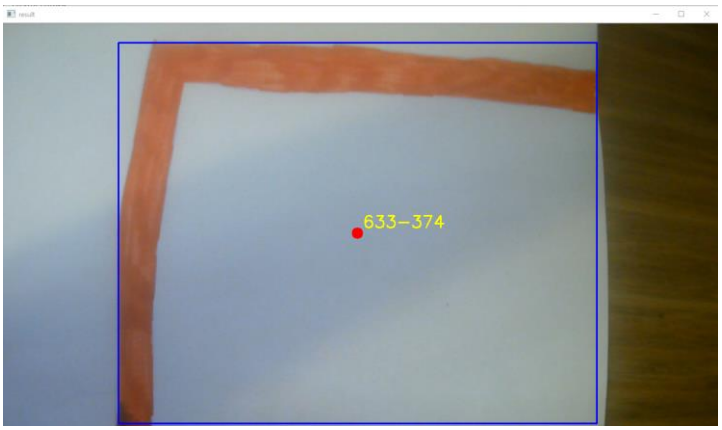

Рисунок 30 – Тестирование Boundary Following алгоритма

На основе этого можно сделать вывод, что цветовой фильтр работает правильно и что Boundary Following алгоритм правильно обнаружил границы искомого объекта.

Наличие посторонних объектов в кадре никак не влияет на точность определения линии маршрута. Это продемонстрировано на рисунке 31.

Рисунок 31 – Поиск линии маршрута при наличии посторонних объектов в кадре

В том случае, если цветовой фильтр настроен некорректно, то синий прямоугольник будет обводить либо другие объекты, либо не будет виден в кадре вовсе.

Далее необходимо протестировать то, как вычисляется поправка на смещение. Для начала стоит уточнить, что ширина линии маршрута в среднем занимает 160 пикселей (величина зависит от расстояния от линии до камеры), а ширина кадра равна 1280 пикселей. Теперь проверим вычисление поправки на смещение в том случае, когда в кадре присутствует только прямая линия. На рисунке 32 есть необходимые данные, а именно: координата центра прямоугольника по оси абсцисс и вычисленная поправка при помощи алгоритма, эта поправка обозначается буквой d на изображении.

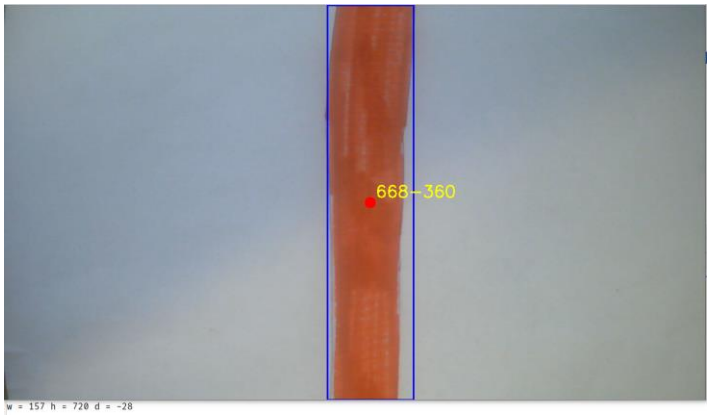

Рисунок 32 – Проверка вычисления поправки на смещение при наличии прямой линии

Подставим значения в формулу 1 и проверим правильность:

$$delta = \frac{1280}{2} - 668 = 640 - 668 = -28$$

Как видно из результатов, значения совпали, значит поправка вычисляется верно.

Проверим аналогичным образом вычисление поправки для случая, когда в кадре виден поворот налево (рисунок 33).

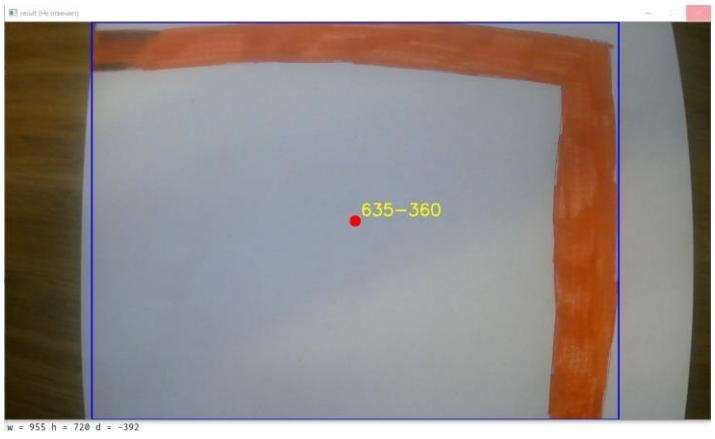

Рисунок 33 – Проверка вычисления поправки при наличии поворота налево

Подставим значения в формулу 2:

$$delta = \frac{1280}{2} - \left(955 - \frac{160}{2} + 157\right) = 640 - 1032 = -392$$

Поправка вычислена правильно, также она получилась отрицательной. Как было сказано раннее, это означает необходимость сместиться вправо, что и видно на рисунке 36.

Теперь проверим вычисление поправки в случае, когда в кадре присутствует поворот направо (рисунок 37). Также подставим значения в формулу 3:

$$delta = \frac{1280}{2} - \left(167 + \frac{160}{2}\right) = 640 - 247 = 393$$

Поправка также была верно вычислена, и она получилась положительной, значит необходимо сместиться влево. Это видно на рисунке 34.

Рисунок 34 – Проверка вычисления поправки при наличии поворота налево

Теперь протестируем алгоритм движения на момент обработки поворотов налево и направо. На рисунках 35 и 36 видны результаты тестов.

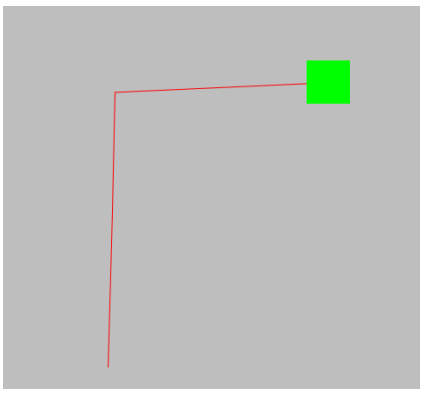

Рисунок 35 – Тестирование поворота направо

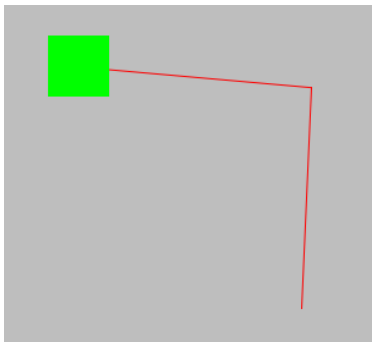

Рисунок 36 – Тестирование поворота налево

Результаты тестов показывают, что алгоритм успешно обрабатывает повороты.

3.2 Анализ результатов тестирования

Проведённое тестирование показало, что разработанный алгоритм движения беспилотного объекта является рабочим и может быть применён на практике.

Алгоритм мгновенно обрабатывает изображение, что, в конечном счёте, позволяет оперативно реагировать на изменение обстановки в кадре. Например, коробка упала на линию маршрута и перегородила дорогу, тогда алгоритм распознает, что пройти вперёд невозможно и остановится.

Стоит отметить, что алгоритм позволяет обнаружить линию маршрута даже в условиях недостаточной освещённости (рисунок 37). В комнате выключены все лампы, а единственный источник света – это монитор. Но даже в таких условиях алгоритм способен найти линию маршрута.

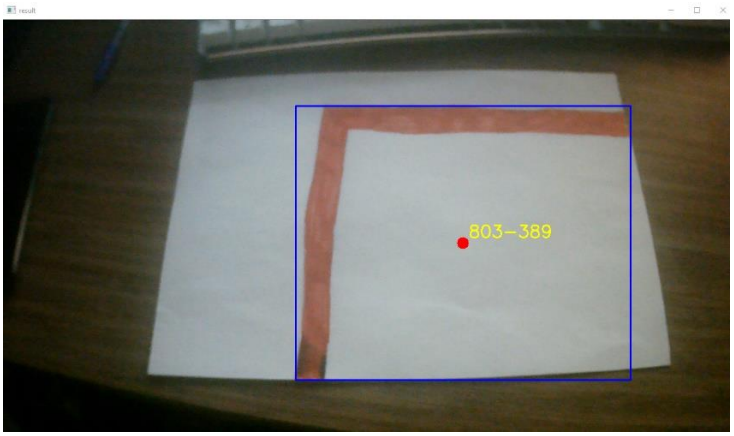

Рисунок 37 – Обнаружение линии в условиях недостаточной освещённости

Также во время тестирования было обнаружено, что программа для моделирования алгоритма движения беспилотного объекта использует от 50 до 60 мегабайт оперативной памяти (рисунок 38). А программа калибровки цветового фильтра использует от 40 до 50 мегабайт оперативной памяти (рисунок 39).

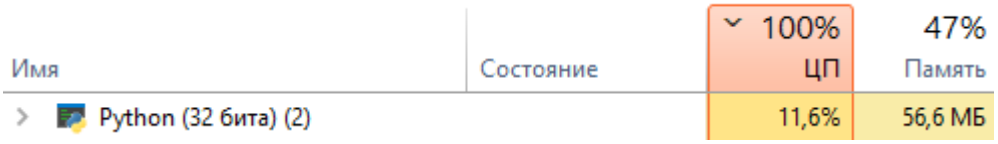

Рисунок 38 – Объём потребляемой памяти во время работы основной программы

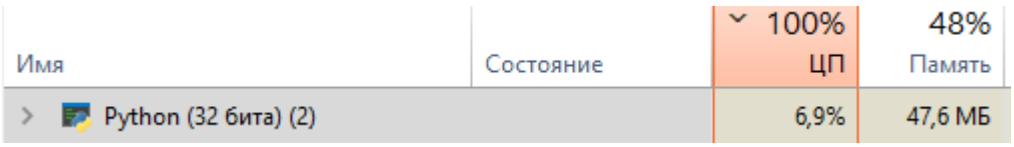

Рисунок 39 – Объём потребляемой памяти во время работы программы для калибровки цветового фильтра

Результаты тестирования показали низкие затраты на вычислительные мощности, а также высокую скорость обработки изображения. Разработанное программное обеспечение можно запускать даже на таких компактных устройствах, как Raspberry Pi 4.

Выводы и результаты по третьему разделу

Таким образом, было выполнено тестирование работоспособности разработанного алгоритма движения беспилотного объекта по линии маршрута на полу. Для тестирования был использован стационарный компьютер, веб-камера Canyon CNS-CWC6N и лист бумаги формата А4 с нарисованной линией красным маркером.

Результаты показали высокую скорость распознавания линии маршрута, при этом от камеры не требуется наличие чёткой картинки и даже не требуется наличие хорошей освещённости от помещения.

Также было выявлено то, что для работы программы необходимо использовать менее 100 мегабайт оперативной памяти.

На основе полученных данных, алгоритм можно признать рабочим.

Заключение

В ходе выполнения выпускной квалификационной работы был разработан алгоритм движения беспилотного объекта, который позволяет перемещаться по линии маршрута на полу, при этом линия может быть любого цвета. Также были разработаны две программы. Первая программа реализует калибровку цветового фильтра. Цветовой фильтр необходим для того, чтобы настроить работу алгоритма на определённый цвет линии маршрута. Полученные данные с программы калибровки необходимо использовать во второй программе. Вторая программа реализует непосредственно сам алгоритм движения и визуализирует его при помощи кроссплатформенного графического интерфейса, предоставляемого библиотекой PyGame.

В процессе выполнения были проанализированы существующие решения беспилотных роботов-погрузчиков в сфере складской логистики, которые предоставляют как отечественные производители, так и зарубежные. Проанализировав полученные данные, были выработаны требования, предъявляемые к беспилотному объекту и его программному обеспечению: беспилотный объект должен быть оборудован камерой с разрешением матрицы не менее 640x480 пикселей, камера должна быть зафиксирована на определённой высоте от пола, а высота не должна меняться в процессе движения; также беспилотный объект должен иметь возможность выполнять код для языка Python версии 3.8 с установленными пакетами pygame, opencv-python, cv2 и numpy.

Разработанный алгоритм позволит уменьшить затраты на внедрение беспилотных объектов, задача которых заключается в перемещении по заранее определённому маршруту, благодаря требованию к наличию только зафиксированной камеры. Такие беспилотные объекты будут значительно дешевле по сравнению с другими аналогами, которым необходимо наличие различных датчиков (например, ультразвуковых).

Список используемой литературы

1. Вордерман, К. Программирование на Python. / К. Вордерман, К. Стили, К. Квигли. - М.: Манн, Иванов и Фербер, 2017. - 346 с

2. Различные цветовые модели и их использование [Электронный ресурс]. – URL: http://tm.spbstu.ru/Различные_цветовые_модели_и_их_использование

3. Саммерфилд, М. Программирование на Python 3. Подробное руководство / М. Саммерфилд. - М.: Символ, 2016. - 608 с.

4. Синтез цвета // Фотокинотехника: Энциклопедия / Гл. ред. Е. А. Иофис. — М.: Советская энциклопедия, 1981. — 447 с.OpenCV на python: цветовой фильтр [Электронный ресурс] – URL: https://robotclass.ru/tutorials/opencv-color-range-filter/

5. События клавиатуры. Урок 4 курса «Pygame. Введение в разработку игр на Python» [Электронный ресурс]. – URL: https://younglinux.info/pygame/key

6. Уэйкерли Дж. Ф. Проектирование цифровых устройств, том 2. — М.: Постмаркет, 2002. С. 620—621.

7. Фильтр цвета с OpenCV [Электронный ресурс]. – URL: http://espressocode.top/filter-color-with-opencv/

8. Ян Эрик Солем Программирование компьютерного зрения на языке Python. / пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2016. – 312 с.

9. Canyon CNS-CWC6N. Technical specification [Электронный ресурс]. – URL: https://canyon.eu/wp-content/uploads/RS12807_CNS-CWC6N-_EN.pdf

10. Drawing graphics primitives – Pygame tutorial 2019 documentation [Электронный ресурс]. – URL: https://pygame.readthedocs.io/en/latest/2_draw/draw.html

11. Joseph Howse. OpenCV Computer Vision with Python. – UK: Livery Place, 2013. – 122 p. – ISBN 978-1-78216-392-3

12. OpenCV на python: цветовой фильтр [Электронный ресурс] – URL: https://robotclass.ru/tutorials/opencv-color-range-filter/

13. OpenCV: Contour Approximation Modes [Электронный ресурс]. – URL: https://docs.opencv.org/master/d3/dc0/group__imgproc__shape.html

14. OpenCV: Retrieval Modes [Электронный ресурс]. – URL: https://docs.opencv.org/master/d3/dc0/group__imgproc__shape.html#ga819779b9857cc2f8601e6526a3a5bc71

15. PyGame documentation [Электронный ресурс]. – URL: https://www.pygame.org/docs/

16. Pygame Mouse Click and Detection – CodersLegacy [Электронный ресурс]. – URL: https://coderslegacy.com/python/pygame-mouse-click/

17. Raspberry Pi 4 Model B specification [Электронный ресурс]. – URL: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/

18. Samyak Datta. Learning OpenCV 3 Application Development. – UK: Livery Place, 2016. – 305 p. – ISBN 978-1-78439-145-4

19. Suzuki, S. and Abe, K., Topological Structural Analysis of Digitized Binary Images by Border Following. CVGIP 30 1, pp 32-46 (1985)

20. Teh, C.H. and Chin, R.T., On the Detection of Dominant Points on Digital Curve. PAMI 11 8, pp 859-872 (1989)

Приложение А

Листинг программы для калибровки цветового фильтра

```
import cv2
import numpy as np

if __name__ == '__main__':
    def nothing(*arg):
        pass

cv2.namedWindow("result")  # создаем главное окно
cv2.namedWindow("settings")  # создаем окно настроек

webcam = cv2.VideoCapture(0)
webcam.set(3, 1280)
webcam.set(4, 720)

# создаем 6 бегунков для настройки начального и конечного цвета фильтра
cv2.createTrackbar('h1', 'settings', 0, 255, nothing)
cv2.createTrackbar('s1', 'settings', 0, 255, nothing)
cv2.createTrackbar('v1', 'settings', 0, 255, nothing)
cv2.createTrackbar('h2', 'settings', 255, 255, nothing)
cv2.createTrackbar('s2', 'settings', 255, 255, nothing)
cv2.createTrackbar('v2', 'settings', 255, 255, nothing)

while True:
    flag, img = webcam.read()
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    # считываем значения бегунков
    h1 = cv2.getTrackbarPos('h1', 'settings')
    s1 = cv2.getTrackbarPos('s1', 'settings')
    v1 = cv2.getTrackbarPos('v1', 'settings')
    h2 = cv2.getTrackbarPos('h2', 'settings')
    s2 = cv2.getTrackbarPos('s2', 'settings')
    v2 = cv2.getTrackbarPos('v2', 'settings')

    # формируем начальный и конечный цвет фильтра
    h_min = np.array((h1, s1, v1), np.uint8)
    h_max = np.array((h2, s2, v2), np.uint8)

    # накладываем фильтр на кадр в модели HSV
    thresh = cv2.inRange(hsv, h_min, h_max)

    cv2.imshow('result', thresh)

    ch = cv2.waitKey(5)
    if ch == 27:
        break

webcam.release()
cv2.destroyAllWindows()
```

Приложение Б

Листинг программы для моделирования разработанного алгоритма движения беспилотного объекта

```
import pygame
import cv2
import numpy as np


# инициализация параметров изображения
MAX_WIDTH = np.int32(1280)
MAX_HEIGHT = np.int32(720)
LINE_WIDTH = 160

# инициализация камеры
cv2.namedWindow("result")
webcam = cv2.VideoCapture(0)
webcam.set(3, MAX_WIDTH)
webcam.set(4, MAX_HEIGHT)

# инициализация pygame
pygame.init()
pygame.display.set_caption("route")
surface = pygame.display.set_mode([MAX_WIDTH, MAX_HEIGHT])
clock = pygame.time.Clock()

# определение цветового фильтра
hsv_min = np.array((0, 84, 116), np.uint8)
hsv_max = np.array((14, 255, 255), np.uint8)

# переменные для отображения траектории
i, j, SIZE = MAX_WIDTH/2, MAX_HEIGHT-50, 50
step, delta, height, angle = 1, 0, 0, 0
dots = np.array([[MAX_WIDTH/2, MAX_HEIGHT-50]])
dots_for_display = np.array([[MAX_WIDTH/2, MAX_HEIGHT-50]])
coords = {"x": 0, "y": 0}
dirs = {0:   [["y", -1], ["x",  1]],
        90:  [["x", -1], ["y", -1]],
        180: [["y",  1], ["x", -1]],
        270: [["x",  1], ["y",  1]]}

MOVING, TIMEOUT_START = False, False
start_ticks = 0

while True:
    # вычисление координат текущей позиции
    currentPos = [i + coords["x"], j + coords["y"]]
    # проверка на момент остановки
    if height <= LINE_WIDTH:
        coords["x"], coords["y"] = 0, 0
        if currentPos not in dots:
            dots = np.vstack((dots, currentPos))
            i, j = currentPos
    elif height > LINE_WIDTH:
        coords[dirs[angle][0][0]] += step * dirs[angle][0][1]

    # смещение по горизонтали
    if abs(delta) > 20:
        diff = step * dirs[angle][1][1] if delta < 0 else -step *
dirs[angle][1][1]
        coords[dirs[angle][1][0]] += diff
```

```
    dots_for_display = dots.copy()
    dots_for_display = np.vstack((dots_for_display, currentPos))

    # отрисовка пройденного маршрута
    surface.fill(pygame.Color("gray"))
    pygame.draw.aalines(surface, pygame.Color("Red"), False,
dots_for_display)
    pygame.draw.rect(surface, pygame.Color("Green"), currentPos + [SIZE,
SIZE])

    # считывание изображение с камеры и применение цветового фильтра
    flag, img = webcam.read()
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    thresh = cv2.inRange(hsv, hsv_min, hsv_max)

    # поиск и фильтрация контуров
    contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)
    filtered = [c for c in contours if cv2.contourArea(c) > 5000]
    if len(filtered) == 1:
        TIMEOUT_START = False
        # вычисление координат левого верхнего угла, ширины и высоты
прямоугольника
        x, y, width, height = cv2.boundingRect(filtered[0])
        # вычисление координат центра прямоугольника
        x_center, y_center = int(x + width / 2), int(y + height / 2)
        # вычисление поправки на смещение по горизонтали
        if width <= LINE_WIDTH:
            delta = int(MAX_WIDTH / 2 - x_center)
        elif width > LINE_WIDTH and x_center < MAX_WIDTH / 2:
            delta = int(MAX_WIDTH / 2 - (width - LINE_WIDTH/2 + x))
        elif width > LINE_WIDTH and x_center > MAX_WIDTH / 2:
            delta = int(MAX_WIDTH / 2 - (x + LINE_WIDTH/2))
        else:
            delta = 0

        # проверка на поворот
        if width > LINE_WIDTH >= height:
            coords["x"], coords["y"] = 0, 0
            if currentPos not in dots:
                dots = np.vstack((dots, currentPos))
                i, j = currentPos
            angle = (angle + 90) % 360 if x_center < MAX_WIDTH / 2 else
(angle - 90) % 360

        # вывод технической информации
        print("w = %d h = %d d = %d a = %d" % (width, height, delta,
int(cv2.contourArea(filtered[0]))))
        # отрисовка контура и его центра
        cv2.rectangle(img, (x, y), (x + width, y + height), (255, 0, 0), 2)
        cv2.circle(img, (x_center, y_center), 10, (0, 0, 255), -1)
        cv2.putText(img, "%d-%d" % (x_center, y_center), (x_center + 10,
y_center - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 255), 2)
    else:
        # проверка на отсутствие контуров в течение пяти секунд
        width, height, delta = 0, 0, 0
        if not TIMEOUT_START:
            start_ticks = pygame.time.get_ticks()
            TIMEOUT_START = True
        else:
```

```
            seconds = (pygame.time.get_ticks() - start_ticks) / 1000
            if seconds > 5:
                print("Timeout")
                break

    cv2.imshow('result', img)

    pygame.display.flip()
    clock.tick(30)

    key = pygame.key.get_pressed()
    if key[pygame.K_ESCAPE]:
        break

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            exit()

        if event.type == pygame.MOUSEBUTTONDOWN:
            MOVING = True
        if event.type == pygame.MOUSEBUTTONUP:
            MOVING = False

        if event.type == pygame.MOUSEMOTION and MOVING:
            dots = np.add(dots, event.rel)
            dots_for_display = np.add(dots_for_display, event.rel)
            i, j = np.add((i, j), event.rel)


webcam.release()
cv2.destroyAllWindows()
```