Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский авиационный институт

(национальный исследовательский университет)»

Отчёт по вычислительной практике

|  |  |  |
| --- | --- | --- |
| Исполнитель, студент группы М3О-214Б-19 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Михалченков В.М. |
| Руководитель, старший преподаватель каф.307 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Максимов А.Н. |

Москва, 2021

Оглавление

[Постановка задачи 3](#_Toc77858376)

[Техническое задание 3](#_Toc77858377)

[1. Введение 3](#_Toc77858378)

[2. Основание для разработки 3](#_Toc77858379)

[3. Назначение разработки 3](#_Toc77858380)

[4. Требования к программе и программному изделию 3](#_Toc77858381)

[5. Требования к программной документации 4](#_Toc77858382)

[6. Стадии и этапы разработки 4](#_Toc77858383)

[Описание алгоритма решения 5](#_Toc77858384)

[Описание программы моделирования окружающей среды 5](#_Toc77858385)

[Описание программы оператора 7](#_Toc77858386)

[Описание алгоритма обработки изображения 9](#_Toc77858387)

[Описание взаимодействия программ 12](#_Toc77858388)

[Результаты работы 14](#_Toc77858389)

[Заключение 16](#_Toc77858390)

[Руководство пользователя 17](#_Toc77858391)

[Приложение А 18](#_Toc77858392)

[Приложение Б 28](#_Toc77858393)

# Постановка задачи

В качестве задания на летнюю вычислительную практику выбрана задача разработки программы для робота, который заталкивает контрастную шайбу в ворота. В соответствии с поставленной задачей разработано техническое задание.

# Техническое задание

## 1. Введение

Техническое задание определяет перечень требований к программе управления оператора, модели робота и его программному обеспечению в соответствии с заданием.

## 2. Основание для разработки

Основанием для разработки послужило задание, выданное на летнюю вычислительную практику: разработка программы для робота, который заталкивает контрастную шайбу в ворота.

## 3. Назначение разработки

Реализация алгоритма, с помощью которого робот сможет затолкнуть контрастную шайбу в ворота, а также программы отслеживания состояния робота и его управления.

## 4. Требования к программе и программному изделию

4.1.Требования к функциональным характеристикам

Программа функционирования робота реализует следующие функции:

- функции поиска и обнаружения цели,

- прямолинейное движение,

- разворот на месте,

- передача данных с камеры оператору.

4.2.Требования к условиям эксплуатации

Система должна эксплуатироваться в помещении при условии высокого уровня освещенности.

4.3.Требования к составу и параметрам технических средств

Колесная база с установленной камерой и модулем управления с поддержкой Wi-Fi, компьютер с установленной программой оператора с поддержкой Wi-Fi.

4.4.Требования к информационной и программной совместимости

Модель должна разрабатываться на языке программирования С#. Модель робота и среды окружения должна разрабатываться в среде разработки Unity.

4.5.Дополнительные специальные требования не предъявлены.

## 5. Требования к программной документации

Программная документация должна включать в себя:

1.Руководство пользователя

2.Программа методики испытаний

## 6. Стадии и этапы разработки

Разработка проекта включает следующие этапы:

1. Определение требований изделия

2. Проектирование программы

3. Разработка программы

4. Тестирование программы

5. Составление документации

На завершающем практическом занятии необходимо представить программу, отчет по практике и исходный код приложения.

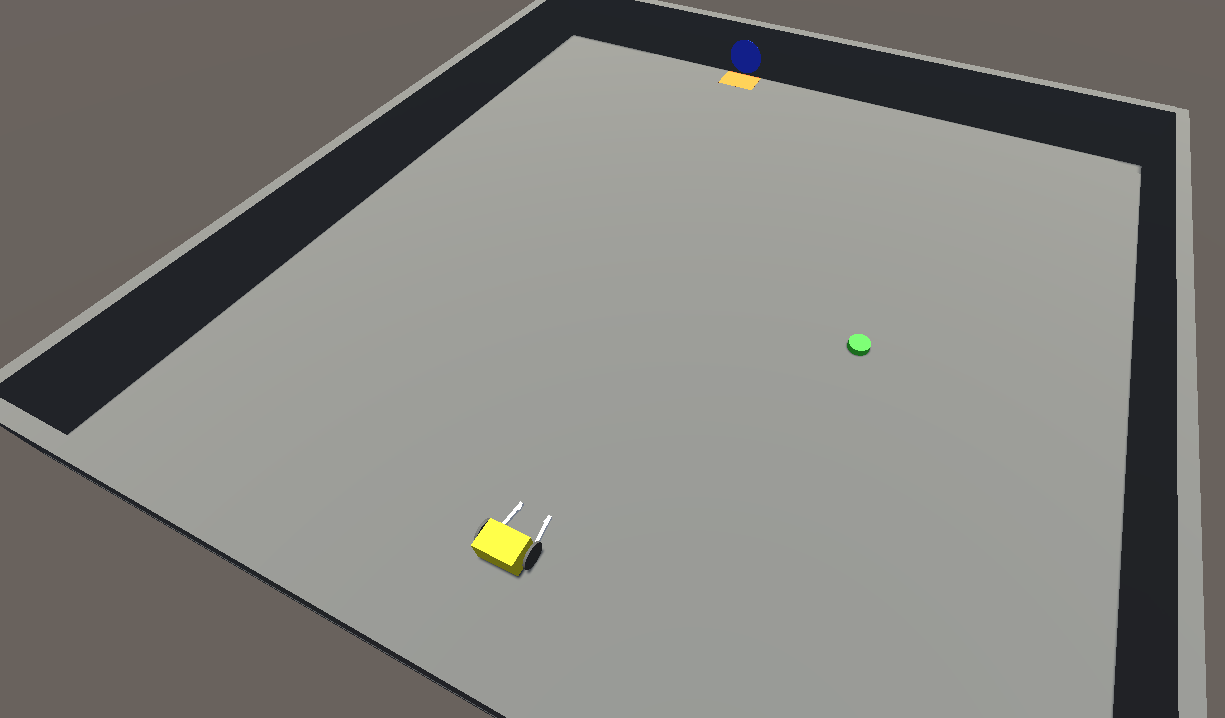
# Описание алгоритма решения

Исходя из поставленной задачи, принято решение о разделении программы на две части: программа симуляции окружающей среды и модели робота и программа оператора. Для передачи сообщений между программой симуляции и программой оператора используется протокол MQTT и брокер Mosquitto.

## Описание программы моделирования окружающей среды

В первой программе симулируется окружение робота и его модель. Программа написана в среде разработки Unity. Окружением робота служит ограниченная по размерам комната, в которой размещен искомый объект — контрастная шайба, область ворот с контрастной меткой и модель робота (Рисунок 1).

Рисунок 1 — Окружение модели робота



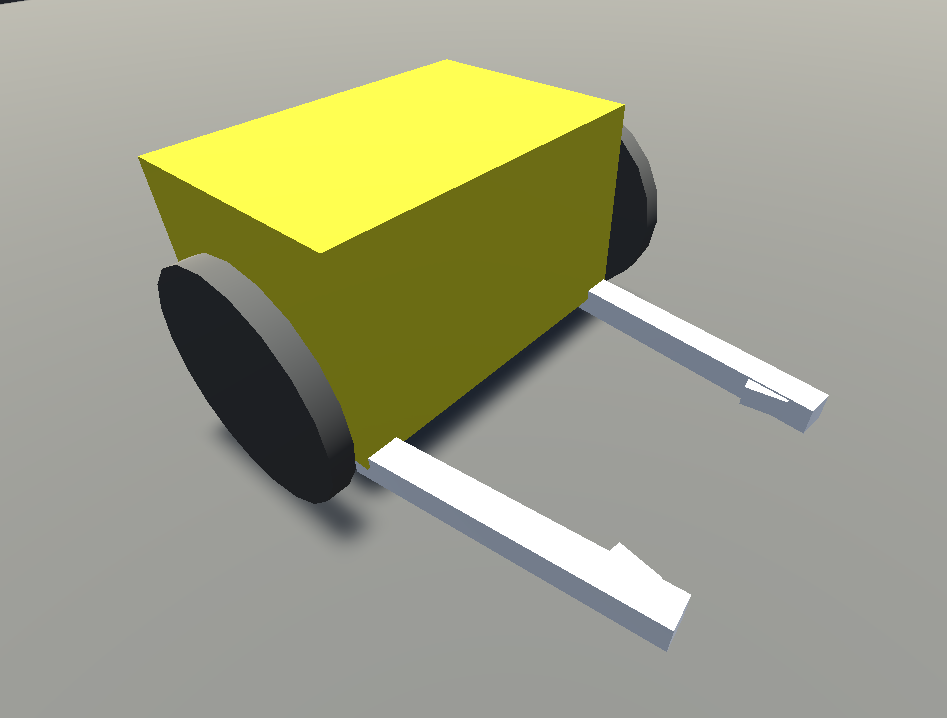
Область комнаты освещена достаточно для того, чтобы получать яркую и контрастную картинку с камеры робота. Модель робота состоит из корпуса в форме параллелепипеда, двух боковых колес и держателей, предназначенных для снижения вероятности потери шайбы при движении робота (Рисунок 2). Для симуляции физически реалистичных колес используется модель Wheel Collider из стандартных физических моделей среды разработки Unity. Использование реалистичной модели колес позволяет более точно воспроизвести поведение реального робота и проводить разработку алгоритма управления учитывая физическое поведение реальных колес. В передней части робота установлена перспективная камера с горизонтальным углом обзора 90°, с помощью которой из среды симуляции получается изображение, которое передается в программу оператора для дальнейшей работы.

Рисунок 2 — Модель робота

Логика управления моделью робота основана на машине состояний. Определены состояния перемещения модели робота: прямолинейное движение вперед и назад, разворот на месте по и против часовой стрелки, остановка. Также определены состояния типа управления: автоматическое и ручное. Переключение между состояниями в автоматическом типе управления происходит согласно командам, получаемым при анализе изображения с камеры робота и приходящим от приложения оператора. Переключение между состояниями в ручном режиме управления происходит посредством команд, приходящих от приложения оператора по нажатию кнопок управления.

При начальном запуске программы симуляции установлен режим управления «Ожидание». Для начала работы необходимо выбрать режим управления в программе оператора.

Автоматическое управление в робота основано на односвязном списке состояний, которые робот последовательно выполняет. Перейти к следующему состоянию в списке возможно только после завершения выполнения текущего состояния. После завершения выполнения всех команд в списке, отправляется сообщение о завершении выполнения текущего списка команд. В любой момент работы робота в конец списка состояний могут быть добавлены новые команды, посредством отправки сообщений из программы оператора.

В режиме ручного управления возможно назначить текущее состояние движения модели робота с помощью нажатия кнопок управления в программе оператора. Команды ручного управления не заносятся в список и выполняются непосредственно после нажатия кнопки управления в программе оператора.

При переходе в автоматический режим модель робота выполнит набор предустановленных команд необходимых для калибровки движения робота: в течение нескольких секунд модель робота выполняет разворот и прямолинейное движение, рассчитывая приблизительную скорость своего передвижения по поверхности, на которой находится.

## Описание программы оператора

Программа оператора представляет собой оконное приложение, с областью вывода картинки с камеры робота и кнопками управления (Рисунок 3). Интерфейс программы оператора написан с помощью библиотеки CSFML (оболочка SFML для .NET) на языке C#.

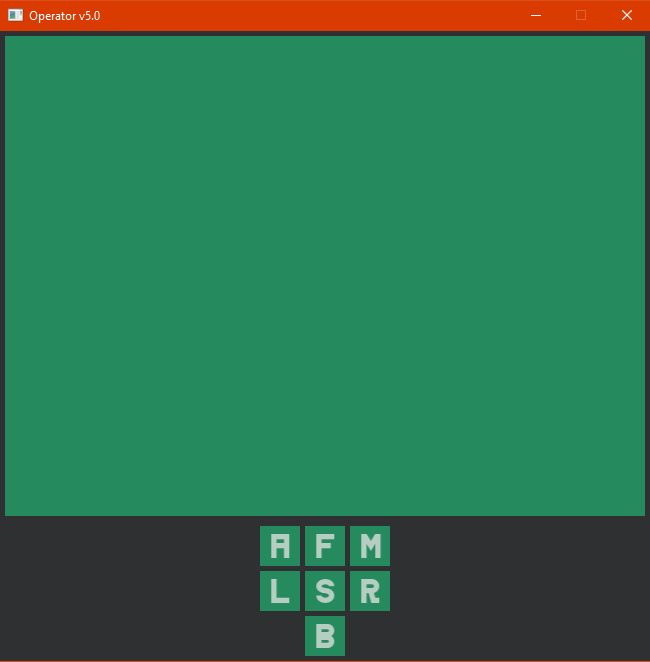
Изображение с камеры робота выводится постоянно с момента подключения робота к брокеру, независимо от режима управления. В нижней части окна приложения находятся кнопки управления. Кнопки с обозначениями «А» и «М» переключают режимы управления на автоматический и ручной соответственно. При выборе ручного режима управления становятся активны кнопки с обозначениями «F» и «B» (прямолинейное движение вперед и назад соответственно), «R» и «L» (разворот на месте в правую и левую стороны соответственно), «P» (остановка).

Рисунок 3 — Приложение оператора

После нажатия кнопки автоматического режима в модели робота будет установлен соответствующий режим и выполнена предустановленная последовательность команд для калибровки движения робота. После чего программа перейдет к алгоритму обработки изображения.

## Описание алгоритма обработки изображения

Обработка изображения производится с помощью библиотеки Emgu CV (оболочка OpenCV для .NET). Алгоритм анализа изображения построен следующим образом: если робот завершил выполнение текущей последовательности команд, то в программе оператора выполнится процесс анализа изображения, в зависимости от результата которого в программу симуляции будет передан набор команд. После выполнения этих команд роботом, он снова передаст сообщение о выполнении всех команд из списка и анализ изображения в программе оператора будет произведен снова.

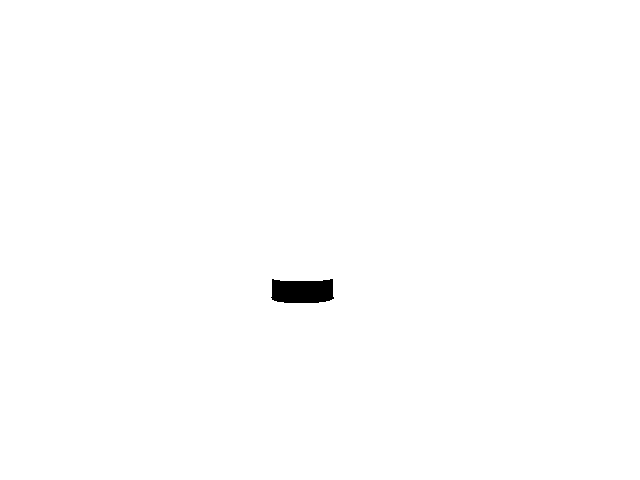
Для поиска объекта на изображении применяется выделение пикселей, принадлежащих диапазону цветов, для этого обрабатываемое изображение переводится в цветовую палитру HSV. На выходе такая функция выдает бинарное изображение (Рисунок 4).

Рисунок 4 — Пример бинарного изображения после выделения пикселей, принадлежащих диапазону значений цвета искомого объекта (шайбы)

Далее такое изображение подается в функцию, выделяющую массив точек контуров, также в этой функции выполняется фильтрация изображения.

Исходя из того, что шайба принята контрастным объектом, предполагается, что в окружении робота не находятся объекты такого же цвета.

После на основании массива точек контуров выделяются ограничивающие эти контуры прямоугольники.

Если количество ограничивающих прямоугольников равно 0 или больше, чем 1, предполагается, что на изображении нет искомого объекта, составляется последовательность команд поворота по часовой стрелке на 45° и паузы в 0.5 секунды и отправляется в программу симуляции, при этом к счетчику неудачных попыток обнаружить объект прибавляется единица, после достижения счетчиком значения равного 10, считается, что искомый объект так и не был найден и управление робота переходит в ручной режим.

Если количество прямоугольников равно 1, предполагается, что в поле зрения камеры робота присутствует искомый объект. Для вычисления угла расположения объекта относительного текущего направления робота производится переход от экранной пиксельной системы координат к UV координатам, затем применяется следующая формула:

где:

X — горизонтальная координата центра ограничивающего прямоугольника искомого объекта, пересчитанная в UV системе координат,

L — ширина ближней плоскости отрисовки перспективной камеры (величина известна из характеристик камеры, использующейся в среде симуляции модели робота),

D — расстояние от точки позиции камеры до ближней плоскости отрисовки (величина известна из характеристик камеры, использующейся в среде симуляции модели робота),

α — полученный угол расположения объекта относительного текущего направления робота, при этом если величина Х будет больше 0.5, то есть центр искомого объекта располагается правее средней линии поля зрения робота, угол будет положительным, и поворот выполнится вправо, иначе — угол отрицательный, поворот влево.

Величина Х вычисляется по следующей формуле:

где:

Rx — значение положения верхнего левого угла ограничивающего прямоугольника выраженное в экранных пиксельных координатах,

Rw — значение ширины ограничивающего прямоугольника выраженное в экранных пиксельных координатах,

W — ширина изображения в пикселях.

После вычисления угла поворота для робота, в программу симуляции будет отправлена последовательность команд: поворот на вычисленный угол, прямолинейное движение вперед на расстояние 1 метр и остановка на 0.5 секунды.

После выполнения этой последовательности команд, робот проведет анализ изображения еще раз. Такая реализация движения к объекту выбрана из-за того, что в симуляции используется физическая модель колес, подача крутящего момента и торможение не происходит мгновенно, из-за наличия подвески корпус робота на поворотах и торможении может заносить в сторону от рассчитанного направления, при движении с шайбой робот также может отклоняться от установленного направления — периодически требуется коррекция направления движения.

Шайба считается захваченной в держатели, если располагается на изображении с камеры в определенной области.

После захвата шайбы в держатели, начинается поиск контрастной метки области ворот по вышеописанному алгоритму, при этом происходит и контроль того, что шайба не выпала из держателей. В случае, если шайба выпадает, алгоритм снова переключится на поиск шайбы.

При достижении области ворот, что контролируется обнаружением контрастной метки на стене над областью ворот, робот разместит в ней шайбу и перейдет в ручной режим.

## Описание взаимодействия программ

Для обмена данными между программами используется библиотека M2MQTT, реализующая протокол MQTT для .NET. Общение выполняется через брокер Mosquitto. Определены следующие топики сообщений:

/image — используется для передачи массива байт декодированного изображения с камеры робота в программу оператора,

/command/mode — используется для сообщений, устанавливающих режим управления моделью робота,

/command/auto — используется для пересылки сообщений с командами для автоматического режима управления,

/command/manual — используется для пересылки сообщений с командами для ручного режима управления,

/state — используется для передачи сообщений, содержащих информацию о завершении выполнения моделью робота набора автоматических команд.

**Структурная схема взаимодействия программ **

# Результаты работы

В результате работы написаны две программы: программа, выполняющая симуляцию модели робота и его окружения в среде разработки Unity.

Ниже показан вид с камеры робота, на котором видны искомая шайба, метка области ворот и габариты держателей робота (Рисунок 5).



Рисунок 5 — Вид с камеры из приложения симуляции

Ниже показано окно приложения оператора с кнопками управления и изображением, транслирующимся с камеры робота (Рисунок 6).



Рисунок 6 — Окно приложения оператора

# Заключение

В процессе выполнения вычислительной практики разработан алгоритм распознавания и перемещения объектов роботом в соответствии с заданием, на его основе написаны программы симуляции среды и модели робота и управления моделью робота. Исходный код программы симуляции представлен в Приложении А, исходный код программы оператора представлен в Приложении Б.

# Руководство пользователя

Для использования данных программ необходима ОС Windows 7 (или более новая), установленный .NET Framework версии 4.0 или более новой.

Для программы оператора необходимы пакеты Emgu CV, CSFML и M2MQTT.

Для программы симуляции необходим пакет M2MQTT.

Для работы программ необходим MQTT брокер. Запуск программ можно производить в любой последовательности. При запуске программы оператора в консоли необходимо ввести адрес брокера, либо ввести пустую строку, если брокер размещен на локальной машине. После подключения к брокеру, с программы оператора можно управлять моделью робота с помощью кнопок управления. Кнопки «A» и «M» — выбор режима управления: автоматический и ручной соответственно. Кнопки с обозначениями «F» и «B» (прямолинейное движение вперед и назад соответственно), «R» и «L» (разворот на месте в правую и левую стороны соответственно), «P» (остановка).

# Приложение А

**using** System.Collections.Generic;

**using** uPLibrary.Networking.M2Mqtt;

**using** uPLibrary.Networking.M2Mqtt.Messages;

**using** UnityEngine;

**using** System;

**using** System.Globalization;

**public** **class** RobotControl : MonoBehaviour

{

**public** **static** Vector3 startPosition;

**public** **static** Quaternion startRotation;

**public** **static** Vector3 currentPosition;

**public** **static** Quaternion currentRotation;

**public** **enum** ControlType

{

WAITING = 0,

AUTO,

MANUAL

}

**public** **static** ControlType controlType { **get**; **set**; } = ControlType.WAITING;

**public** **enum** FindingStates

{

FINDING\_PUCK = 0,

FINDING\_GATE = 1

}

**public** **enum** Commands

{

PAUSE = 0,

CHECK\_ROTATION = 1,

CHECK\_MOVING = 2,

STR\_M = 3,

ROT = 4,

STR\_M\_F = 5,

STR\_M\_B = 6,

ROT\_R = 7,

ROT\_L = 8,

END\_OF\_STATE\_SEQUENCE = 9,

IN\_SEQUENCE = 10,

}

**public** **static** Commands state { **get**; **set**; }

**public** MQTTClient client;

**public** AutoPilot auto;

**public** Manual manual;

**public** WheelController whellController;

**public** RobotCamera robotCamera;

**void** Awake()

{

client = **new** MQTTClient();

client.CreateClient();

auto = **new** AutoPilot();

auto.GetClient(client);

manual = **new** Manual();

whellController = **new** WheelController();

GetWheels();

robotCamera = **new** RobotCamera();

robotCamera.GetClient(client);

GetCamera();

SetStartPosition();

}

**void** FixedUpdate()

{

UpdateCurrentPosition();

**if** (controlType == ControlType.MANUAL)

{

manual.DoAction();

}

**if** (controlType == ControlType.AUTO)

{

auto.DoAction();

}

}

**private** **void** LateUpdate()

{

robotCamera.GetFrame();

}

**public** **class** MQTTClient

{

**private** **static** **string** brokerAdress { **get**; **set**; } = "localhost";

**public** **static** **string** imageTopic { **get**; **private** **set**; } = "/image";

**public** **static** **string** autoTopic { **get**; **private** **set**; } = "/command/auto";

**public** **static** **string** manualTopic { **get**; **private** **set**; } = "/command/manual";

**public** **static** **string** modeTopic { **get**; **private** **set**; } = "/command/mode";

**public** **static** **string** stateTopic { **get**; **private** **set**; } = "/state";

**private** NumberFormatInfo formatInfo = **new** NumberFormatInfo() { NumberDecimalSeparator = "." };

**private** **char**[] separators = **new** **char**[] { ';' };

**public** MqttClient client { **get**; **private** **set**; }

**public** **void** CreateClient()

{

client = **new** MqttClient(brokerAdress);

client.MqttMsgPublishReceived += Receive;

client.Connect(Guid.NewGuid().ToString());

client.Subscribe(**new** **string**[] { "#" }, **new** **byte**[] { MqttMsgBase.QOS\_LEVEL\_EXACTLY\_ONCE });

}

**private** **void** Receive(**object** sender, MqttMsgPublishEventArgs e)

{

**if** (e.Topic == autoTopic)

{

**string** Command = "";

**foreach** (**var** item **in** e.Message)

{

Command += Convert.ToChar(item);

}

**string**[] splitCommand = Command.Split(separators, StringSplitOptions.RemoveEmptyEntries);

Debug.Log(Command);

AutoPilot.AddCommandToStateSequence((**int**.Parse(splitCommand[0]), **float**.Parse(splitCommand[1], formatInfo)));

}

**if** (e.Topic == manualTopic)

{

**string** Command = "";

**foreach** (**var** item **in** e.Message)

{

Command += Convert.ToChar(item);

}

Manual.SetCommand(**int**.Parse(Command));

Debug.Log("Manual Mode " + (**int**.Parse(Command)).ToString());

}

**if** (e.Topic == modeTopic)

{

**string** Command = "";

**foreach** (**var** item **in** e.Message)

{

Command += Convert.ToChar(item);

}

controlType = (ControlType)**int**.Parse(Command);

Debug.Log("Command Mode " + controlType.ToString());

}

}

**public** **void** PublishState(**string** state)

{

**byte**[] message = **new** **byte**[state.Length];

**for** (**int** i = 0; i < state.Length; i++)

{

message[i] = Convert.ToByte(state[i]);

}

client.Publish(stateTopic, message, MqttMsgBase.QOS\_LEVEL\_EXACTLY\_ONCE, **false**);

}

**public** **void** PublishImage(**byte**[] message)

{

client.Publish(imageTopic, message, MqttMsgBase.QOS\_LEVEL\_EXACTLY\_ONCE, **false**);

}

}

**public** **class** RobotCamera

{

**public** **static** Camera camera { **get**; **set**; }

**private** **static** **float** captureTime { **get**; **set**; } = 0.2f;

**public** **static** **int** textureWidth { **get**; **private** **set**; } = 640;

**private** **float** time = 0;

**private** MQTTClient cachedClient;

**public** **void** GetClient(MQTTClient \_client)

{

cachedClient = \_client;

}

**public** **void** GetFrame()

{

**if** ((time += Time.deltaTime) > captureTime)

{

**var** RT = **new** RenderTexture(textureWidth, (textureWidth \* 3) / 4, 24);

RenderTexture.active = RT;

**var** image = **new** Texture2D(RT.width, RT.height, TextureFormat.RGB24, **false**);

camera.targetTexture = RT;

camera.Render();

image.ReadPixels(**new** Rect(0, 0, textureWidth, (textureWidth \* 3) / 4), 0, 0);

image.Apply();

**byte**[] bytes = image.EncodeToPNG();

RenderTexture.active = **null**;

camera.targetTexture = **null**;

camera.targetDisplay = 1;

cachedClient.PublishImage(bytes);

time = 0;

Destroy(RT);

Destroy(image);

}

}

}

**private** **void** SetStartPosition()

{

startPosition = transform.position;

startRotation = transform.rotation;

}

**private** **void** UpdateCurrentPosition()

{

currentPosition = transform.position;

currentRotation = transform.rotation;

}

**public** **class** WheelController

{

**public** **static** WheelCollider leftWheel;

**public** **static** WheelCollider rightWheel;

**private** **static** **float** maxMotorTorque { **get**; **set**; } = 0.6f;

**private** **static** **float** maxBrakeTorque { **get**; **set**; } = 300f;

**private** **static** **float** rotationFactor { **get**; **set**; } = 8f;

**public** **static** **void** BrakesOn()

{

leftWheel.brakeTorque = maxBrakeTorque;

rightWheel.brakeTorque = maxBrakeTorque;

}

**public** **static** **void** BrakesOff()

{

leftWheel.brakeTorque = 0f;

rightWheel.brakeTorque = 0f;

}

**public** **static** **void** ForwardMotorTorque()

{

leftWheel.motorTorque = maxMotorTorque;

rightWheel.motorTorque = maxMotorTorque;

}

**public** **static** **void** BackwardMotorTorque()

{

leftWheel.motorTorque = -maxMotorTorque;

rightWheel.motorTorque = -maxMotorTorque;

}

**public** **static** **void** RightRotateTorque()

{

leftWheel.motorTorque = maxMotorTorque / rotationFactor;

rightWheel.motorTorque = -maxMotorTorque / rotationFactor;

}

**public** **static** **void** LeftRotateTorque()

{

leftWheel.motorTorque = -maxMotorTorque / rotationFactor;

rightWheel.motorTorque = maxMotorTorque / rotationFactor;

}

}

**public** **void** GetCamera()

{

RobotCamera.camera = GetComponentInChildren<Camera>();

}

**public** **void** GetWheels()

{

WheelCollider[] wheels = GetComponentsInChildren<WheelCollider>();

WheelController.leftWheel = wheels[0];

WheelController.rightWheel = wheels[1];

}

**public** **class** AutoPilot

{

**public** **static** List<(Commands, **float**)> stateSequence = **new** List<(Commands, **float**)>(); //the sequence of states that the robot passes through

**public** **int** listIterator { **get**; **private** **set**; }

**public** **float** commandValue { **get**; **private** **set**; }

**public** **bool** inAction { **get**; **private** **set**; }

**private** **static** **float** checkTime { **get**; **set**; } = 0;

**private** **static** **float** rotationAnglePerSecond { **get**; **set**; } = 0;

**private** **static** **float** distancePerSecond { **get**; **set**; } = 0;

**private** MQTTClient cachedClient;

**public** **delegate** **void** Action(**float** value);

**public** Action action;

**public** AutoPilot()

{

listIterator = 0;

AddCommandToStateSequence((0, 1f));

AddCommandToStateSequence((1, 3f));

AddCommandToStateSequence((0, 2f));

AddCommandToStateSequence((2, 3f));

AddCommandToStateSequence((0, 1f));

}

**public** **static** **void** AddCommandToStateSequence((**int**, **float**) \_command)

{

stateSequence.Add(((Commands)\_command.Item1, \_command.Item2));

}

**public** **void** DoAction()

{

**if** (!inAction)

{

**if** (listIterator < stateSequence.Count)

{

state = (stateSequence[listIterator].Item1);

commandValue = stateSequence[listIterator].Item2;

listIterator++;

**switch** (state)

{

**case** Commands.PAUSE:

{

action = Pause;

**break**;

}

**case** Commands.CHECK\_ROTATION:

{

action = CheckRotation;

**break**;

}

**case** Commands.CHECK\_MOVING:

{

action = CheckMoving;

**break**;

}

**case** Commands.STR\_M:

{

action = StraightMove;

**break**;

}

**case** Commands.ROT:

{

action = Rotation;

**break**;

}

**default**:

**break**;

}

}

}

**if** (!inAction && listIterator >= stateSequence.Count)

{

cachedClient.PublishState(((**int**)Commands.END\_OF\_STATE\_SEQUENCE).ToString());

}

action(commandValue);

}

**public** **void** CheckRotation(**float** checkTimeValue)

{

**if** (state == Commands.CHECK\_ROTATION)

{

inAction = **true**;

**if** (checkTime > checkTimeValue)

{

WheelController.BrakesOn();

rotationAnglePerSecond = Math.Abs(startRotation.eulerAngles.y - currentRotation.eulerAngles.y) / checkTime;

checkTime = 0;

inAction = **false**;

AddCommandToStateSequence((0, 1f));

}

**else**

{

WheelController.BrakesOff();

WheelController.RightRotateTorque();

checkTime += Time.fixedDeltaTime;

}

}

}

**public** **void** CheckMoving(**float** checkTimeValue)

{

**if** (state == Commands.CHECK\_MOVING)

{

inAction = **true**;

**if** (checkTime > checkTimeValue)

{

WheelController.BrakesOn();

distancePerSecond = (startPosition - currentPosition).magnitude / checkTime;

checkTime = 0;

inAction = **false**;

AddCommandToStateSequence((0, 1f));

}

**else**

{

WheelController.BrakesOff();

WheelController.ForwardMotorTorque();

checkTime += Time.fixedDeltaTime;

}

}

}

**private** **void** Pause(**float** pauseTime)

{

**if** (state == Commands.PAUSE)

{

inAction = **true**;

**if** (checkTime > pauseTime)

{

inAction = **false**;

checkTime = 0;

}

**else**

{

checkTime += Time.fixedDeltaTime;

}

}

}

**public** **void** Rotation(**float** rotationAngleValue)

{

**if** (state == Commands.ROT)

{

inAction = **true**;

**if** (checkTime > Math.Abs(rotationAngleValue) / rotationAnglePerSecond)

{

WheelController.BrakesOn();

AddCommandToStateSequence((0, 1f));

checkTime = 0;

inAction = **false**;

}

**else**

{

WheelController.BrakesOff();

**if** (rotationAngleValue < 0)

{

WheelController.LeftRotateTorque();

}

**else**

{

WheelController.RightRotateTorque();

}

checkTime += Time.fixedDeltaTime;

}

}

}

**public** **void** StraightMove(**float** distanceValue)

{

**if** (state == Commands.STR\_M)

{

inAction = **true**;

**if** (checkTime > Math.Abs(distanceValue) / distancePerSecond)

{

WheelController.BrakesOn();

AddCommandToStateSequence((0, 1f));

checkTime = 0;

inAction = **false**;

}

**else**

{

WheelController.BrakesOff();

**if** (distanceValue < 0)

{

WheelController.BackwardMotorTorque();

}

**else**

{

WheelController.ForwardMotorTorque();

}

checkTime += Time.fixedDeltaTime;

}

}

}

**public** **void** GetClient(MQTTClient client)

{

cachedClient = client;

}

}

**public** **class** Manual

{

**public** **delegate** **void** Action();

**public** Action action;

**private** **static** Commands state { **get**; **set**; } = Commands.PAUSE;

**public** **void** DoAction()

{

**switch** (state)

{

**case** Commands.PAUSE:

{

action = Pause;

**break**;

}

**case** Commands.STR\_M\_F:

{

action = ForwardMove;

**break**;

}

**case** Commands.STR\_M\_B:

{

action = BackwardMove;

**break**;

}

**case** Commands.ROT\_L:

{

action = RotateLeft;

**break**;

}

**case** Commands.ROT\_R:

{

action = RotateRight;

**break**;

}

**default**:

**break**;

}

action();

}

**public** **static** **void** SetCommand(**int** stateNum)

{

state = (Commands)stateNum;

}

**public** **void** Pause()

{

WheelController.BrakesOn();

}

**public** **void** RotateLeft()

{

WheelController.BrakesOff();

WheelController.LeftRotateTorque();

}

**public** **void** RotateRight()

{

WheelController.BrakesOff();

WheelController.RightRotateTorque();

}

**public** **void** ForwardMove()

{

WheelController.BrakesOff();

WheelController.ForwardMotorTorque();

}

**public** **void** BackwardMove()

{

WheelController.BrakesOff();

WheelController.BackwardMotorTorque();

}

}

}

# Приложение Б

**using** System;

**using** System.Collections.Generic;

**using** uPLibrary.Networking.M2Mqtt;

**using** uPLibrary.Networking.M2Mqtt.Messages;

**using** SFML.Graphics;

**using** SFML.System;

**using** SFML.Window;

**using** Emgu.CV;

**using** Emgu.CV.Util;

**using** Emgu.CV.Structure;

**using** System.Drawing;

**namespace** window\_core

{

**class** Program

{

**public** **static** MainWindow window = **new**();

**static** **void** Main(**string**[] args)

{

window.InitializeWindow(650, 630);

Console.WriteLine("Press ESC key to close window");

window.RunMainCycle();

Console.WriteLine("Successful exit");

}

}

**class** MQTTClient

{

**public** MqttClient client { **get**; **private** **set**; }

**private** **string** brokerHostName { **get**; **set**; } = "localhost";

**public** **static** **string** imageTopic { **get**; **private** **set**; } = "/image";

**public** **static** **string** autoTopic { **get**; **private** **set**; } = "/command/auto";

**public** **static** **string** manualTopic { **get**; **private** **set**; } = "/command/manual";

**public** **static** **string** modeTopic { **get**; **private** **set**; } = "/command/mode";

**public** **static** **string** stateTopic { **get**; **private** **set**; } = "/state";

**public** **static** **byte**[] recivedImageBytes { **get**; **private** **set**; }

**public** **static** **bool** isNewImageBytes { **get**; **set**; }

**public** **void** InitializeClient()

{

Console.WriteLine("\tEnter brokerHostName in format X.X.X.X or empty for localhost");

**string** enterLine = Console.ReadLine();

**if** (!(enterLine.Length == 0)) { brokerHostName = enterLine; }

client = **new** MqttClient(brokerHostName);

**try**

{

client.Connect(Guid.NewGuid().ToString());

}

**catch** (uPLibrary.Networking.M2Mqtt.Exceptions.MqttConnectionException e)

{

Console.WriteLine(e.Message + " : check server state");

Console.WriteLine();

Environment.Exit(0);

}

Console.WriteLine("\t# topic");

client.Subscribe(**new** **string**[] { "#" }, **new** **byte**[] { MqttMsgBase.QOS\_LEVEL\_EXACTLY\_ONCE });

client.MqttMsgPublishReceived += ReciveMessage;

}

**public** **void** PublishCommand(List<(**string**, **string**)> commandSequence)

{

**foreach** (**var** command **in** commandSequence)

{

**byte**[] commandToPublish = **new** **byte**[command.Item1.Length];

**for** (**int** i = 0; i < command.Item1.Length; i++)

{

commandToPublish[i] = Convert.ToByte(command.Item1[i]);

}

client.Publish(command.Item2, commandToPublish, MqttMsgBase.QOS\_LEVEL\_EXACTLY\_ONCE, **false**);

}

}

**private** **void** ReciveMessage(**object** sender, MqttMsgPublishEventArgs e)

{

**if** (e.Topic == imageTopic)

{

recivedImageBytes = e.Message;

isNewImageBytes = **true**;

}

**if** (e.Topic == stateTopic)

{

**string** state = "";

**foreach** (**var** item **in** e.Message)

{

state += Convert.ToChar(item);

}

RecoObject.command = (Commands)**int**.Parse(state);

}

}

}

**class** MainWindow

{

**public** **static** MQTTClient mqttClient = **new**();

**public** **static** ImageShow image = **new**();

**public** **static** Puck puck = **new** Puck(**new** Hsv(75, 100, 95), **new** Hsv(95, 220, 255), "puck");

**public** **static** Gate gate = **new** Gate(**new** Hsv(3, 175, 115), **new** Hsv(6, 255, 230), "gate");

**public** RenderWindow window;

**public** Image icon = **null**;

**public** VideoMode mode;

**private** **string** windowTitle { **get**; **set**; } = "Operator v5.0";

**private** **static** SFML.Graphics.Color backgroundColor { **get**; **set**; } = **new** SFML.Graphics.Color(47, 48, 50);

**private** List<Button> buttons = **new** List<Button>();

**private** **void** AddButtons()

{

buttons.Add(**new** Button(**new** Vector2f(305, 495), Commands.STR\_M\_F, "F"));

buttons.Add(**new** Button(**new** Vector2f(305, 585), Commands.STR\_M\_B, "B"));

buttons.Add(**new** Button(**new** Vector2f(305, 540), Commands.PAUSE, "S"));

buttons.Add(**new** Button(**new** Vector2f(260, 540), Commands.ROT\_L, "L"));

buttons.Add(**new** Button(**new** Vector2f(350, 540), Commands.ROT\_R, "R"));

buttons.Add(**new** Button(**new** Vector2f(260, 495), Commands.AUTO\_MODE, "A"));

buttons.Add(**new** Button(**new** Vector2f(350, 495), Commands.MANUAL\_MODE, "M"));

}

**private** **void** AddButtonsToDraw()

{

**if** (buttons.Count != 0)

{

**foreach** (**var** button **in** buttons)

{

window.Draw(button.buttonBody);

window.Draw(button.buttonText);

}

}

}

**public** **void** InitializeWindow(**uint** x, **uint** y)

{

mode = **new** VideoMode(x, y);

window = **new** RenderWindow(mode, windowTitle, Styles.Close);

window.KeyPressed += WindowKeyPressed;

window.MouseButtonPressed += WindowMouseButtonPressed;

window.Closed += WindowClosed;

AddButtons();

mqttClient.InitializeClient();

}

**private** **void** WindowClosed(**object** sender, EventArgs e)

{

**var** window = (Window)sender;

window.Close();

}

**public** **void** RunMainCycle()

{

**while** (window.IsOpen)

{

**if** (MQTTClient.isNewImageBytes)

{

**if** (RecoObject.command == Commands.END\_OF\_STATE\_SEQUENCE)

{

List<(**string**, **string**)> commands;

**if** (!puck.caught)

{

puck.CreateImage(MQTTClient.recivedImageBytes);

puck.Recognize();

commands = puck.FollowTo();

puck.CleanAllLists();

}

**else**

{

gate.CreateImage(MQTTClient.recivedImageBytes);

gate.Recognize();

**if** (gate.caught)

{

commands = gate.Stop();

}

**else**

{

commands = gate.FollowTo();

}

gate.CleanAllLists();

}

mqttClient.PublishCommand(commands);

commands.Clear();

}

image.UpdateTexture(MQTTClient.recivedImageBytes);

RecoObject.command = Commands.IN\_SEQUENCE;

}

window.Clear(backgroundColor);

window.Draw(image.sprite);

AddButtonsToDraw();

window.DispatchEvents();

window.Display();

}

}

**private** **void** WindowMouseButtonPressed(**object** sender, MouseButtonEventArgs e)

{

**foreach** (**var** button **in** buttons)

{

button.Action(button.IsPressed(e.X, e.Y), mqttClient);

**if** (button.IsPressed(e.X, e.Y))

{

Console.WriteLine("\tButton " + button.command.ToString());

}

}

}

**private** **void** WindowKeyPressed(**object** sender, KeyEventArgs e)

{

**var** window = (Window)sender;

**if** (e.Code == Keyboard.Key.Escape)

{

window.Close();

}

}

}

**public** **enum** Commands

{

PAUSE = 0,

AUTO\_MODE = 1,

MANUAL\_MODE = 2,

STR\_M\_F = 5,

STR\_M\_B = 6,

ROT\_R = 7,

ROT\_L = 8,

END\_OF\_STATE\_SEQUENCE = 9,

IN\_SEQUENCE = 10,

}

**public** **enum** FindingStates

{

FINDING\_PUCK = 0,

FINDING\_GATE = 1

}

**class** Button

{

**public** RectangleShape buttonBody { **get**; **private** **set**; }

**public** Text buttonText { **get**; **private** **set**; }

**private** **static** Font font { **get**; **set**; }

**private** **static** **uint** caracterSize { **get**; **set**; } = 30;

**private** **static** SFML.Graphics.Color textColor { **get**; **set**; } = **new** SFML.Graphics.Color(229, 228, 226, 192);

**private** Vector2f buttonPosition { **get**; **set**; }

**private** **static** **float** caracterSizeRatio { **get**; **set**; } = 2.8f;

**private** **static** Vector2f buttonSize { **get**; **set**; } = **new** Vector2f(40, 40);

**private** **static** SFML.Graphics.Color buttonColor { **get**; **set**; } = **new** SFML.Graphics.Color(34, 168, 109, 192);

**public** Commands command { **get**; **private** **set**; }

**public** **float** value { **get**; **set**; }

**static** Button()

{

font = **new** Font("AC\_Thermes\_Solid.ttf");

}

**public** Button(Vector2f position, Commands commandType, **string** text)

{

command = commandType;

buttonPosition = position;

buttonBody = **new** RectangleShape(buttonSize);

buttonBody.FillColor = buttonColor;

buttonBody.Position = buttonPosition;

buttonBody.Size = buttonSize;

buttonText = **new** Text(text, font, caracterSize);

**var** textPositionOffset = **new** Vector2f(buttonPosition.X + buttonSize.X / 2 - caracterSize / caracterSizeRatio, buttonPosition.Y + 2);

buttonText.Position = textPositionOffset;

buttonText.FillColor = textColor;

}

**public** **void** Action(**bool** isPressed, MQTTClient client)

{

**if** (isPressed)

{

List<(**string**, **string**)> commands = **new** List<(**string**, **string**)>();

**if** (command == Commands.AUTO\_MODE || command == Commands.MANUAL\_MODE)

{

commands.Add((((**int**)command).ToString(), MQTTClient.modeTopic));

client.PublishCommand(commands);

}

**else**

{

commands.Add((((**int**)command).ToString(), MQTTClient.manualTopic));

client.PublishCommand(commands);

}

commands.Clear();

}

}

**public** **bool** IsPressed(**int** x, **int** y)

{

**return** ((x > buttonPosition.X && x < buttonPosition.X + buttonSize.X) &&

(y > buttonPosition.Y && y < buttonPosition.Y + buttonSize.Y)) ? **true** : **false**;

}

}

**class** ImageShow

{

**public** Image image { **get**; **private** **set**; }

**public** Texture texture { **get**; **private** **set**; }

**public** Sprite sprite { **get**; **private** **set**; }

**public** ImageShow()

{

image = **new** Image(640, 480, **new** SFML.Graphics.Color(34, 168, 109, 192));

texture = **new** Texture(image);

sprite = **new** Sprite(texture);

sprite.Position = **new** Vector2f(5, 5);

}

**public** **void** UpdateTexture(**byte**[] recived)

{

**if** (recived != **null**)

{

image = **new** Image(recived);

texture.Update(image);

sprite.Texture = texture;

}

MQTTClient.isNewImageBytes = **false**;

}

}

**public** **abstract** **class** RecoObject

{

**public** Image<Hsv, **byte**> image { **get**; **set**; } = **null**;

**public** Image<Gray, **byte**> rangedImage { **get**; **set**; } = **null**;

**public** Hsv lower { **get**; **set**; } = **new** Hsv();

**public** Hsv upper { **get**; **set**; } = **new** Hsv();

**public** VectorOfVectorOfPoint contours { **get**; **set**; } = **new** VectorOfVectorOfPoint();

**public** **int**[,] hierarchy { **get**; **set**; } = **new** **int**[3, 3];

**public** List<Rectangle> rectangles { **get**; **set**; } = **new** List<Rectangle>();

**public** **string** name { **get**; **set**; }

**private** **int** noResult { **get**; **set**; } = 0;

**private** **int** maxNoResult { **get**; **set**; } = 10;

**public** **bool** caught { **get**; **set**; }

**public** **static** Commands command { **get**; **set**; } = Commands.IN\_SEQUENCE;

**public** RecoObject(Hsv low, Hsv up, **string** name)

{

lower = low;

upper = up;

**this**.name = name;

}

**public** **abstract** **bool** Catch();

**public** **void** CreateImage(**byte**[] received)

{

Mat mat = **new** Mat();

CvInvoke.Imdecode(received, Emgu.CV.CvEnum.ImreadModes.Unchanged, mat);

CvInvoke.CvtColor(mat, mat, Emgu.CV.CvEnum.ColorConversion.Rgb2HsvFull);

image = mat.ToImage<Hsv, **byte**>();

}

**public** **void** Recognize()

{

rangedImage = image.InRange(lower, upper);

hierarchy = CvInvoke.FindContourTree(rangedImage, contours, Emgu.CV.CvEnum.ChainApproxMethod.ChainApproxSimple);

**for** (**int** i = 0; i < contours.Size; ++i)

{

**if** (0 <= hierarchy[i, 3])

{

**continue**;

}

rectangles.Add(CvInvoke.BoundingRectangle(contours[i]));

}

}

**public** List<(**string**, **string**)> FollowTo()

{

**if** (noResult > maxNoResult)

{

List<(**string**, **string**)> commands = **new** List<(**string**, **string**)>();

commands.Add(("0;0.5", MQTTClient.autoTopic));

commands.Add(("2", MQTTClient.modeTopic));

Console.WriteLine("Объект не найден, выполнен переход в ручной режим");

**return** commands;

}

**else**

{

**if** (rectangles.Count == 1)

{

**if** (Catch())

{

List<(**string**, **string**)> commands = **new** List<(**string**, **string**)>();

caught = **true**;

commands.Add(("0;0.5", MQTTClient.autoTopic));

commands.Add(("3;0.35", MQTTClient.autoTopic));

Console.WriteLine("Захватил " + name);

**return** commands;

}

**else**

{

List<(**string**, **string**)> commands = **new** List<(**string**, **string**)>();

caught = **false**;

**double** xPos = Convert.ToDouble((rectangles[0].X + rectangles[0].Width / 2)) / Convert.ToDouble(image.Width);

**double** angleToTurn = Convert.ToDouble(Math.Atan((xPos - 0.5d) \* 0.44d / 0.22d) \* 180d / Math.PI);

commands.Add(("4;" + (Math.Round(angleToTurn)).ToString(), MQTTClient.autoTopic));

commands.Add(("0;0.5", MQTTClient.autoTopic));

commands.Add(("3;1", MQTTClient.autoTopic));

Console.WriteLine("Распознан " + name + ", поворот на " + (Math.Round(angleToTurn)).ToString());

**return** commands;

}

}

**else**

{

List<(**string**, **string**)> commands = **new** List<(**string**, **string**)>();

commands.Add(("0;0.5", MQTTClient.autoTopic));

commands.Add(("4;45", MQTTClient.autoTopic));

noResult++;

Console.WriteLine("Не распознал " + name + ",продолжаю поиск");

**return** commands;

}

}

}

**public** **void** CleanAllLists()

{

contours.Clear();

rectangles.Clear();

}

}

**public** **class** Puck : RecoObject

{

**public** Puck(Hsv low, Hsv up, **string** name) : **base**(low, up, name) { }

**public** **override** **bool** Catch()

{

**if** ((**double**)rectangles[0].Y / image.Height > 0.75 && ((rectangles[0].X + rectangles[0].Width / 2) - image.Width / 2) < 0.1) { **return** **true**; }

**else** { **return** **false**; }

}

}

**public** **class** Gate : RecoObject

{

**public** Gate(Hsv low, Hsv up, **string** name) : **base**(low, up, name) { }

**public** **override** **bool** Catch()

{

**if** (((**double**)rectangles[0].Height / image.Height) > 0.55) { **return** **true**; }

**else** { **return** **false**; }

}

**public** List<(**string**, **string**)> Stop()

{

List<(**string**, **string**)> commands = **new** List<(**string**, **string**)>();

commands.Add(("3;-2", MQTTClient.autoTopic));

commands.Add(("2", MQTTClient.modeTopic));

**return** commands;

}

}

}