Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Отчёт

по лабораторной работе №1

Дисциплина: Техническое зрение

Тема: Моделирование движения робота с использованием библиотеки OpenCV

Студент гр. 3331506/70401 Паньков И.С.

Преподаватель Варлашин В.В.

« »_____2020 г.

Санкт-Петербург

2020

Цель работы — знакомство с возможностями библиотеки OpenCV.

З а д а н и е

Разработать класс, предоставляющий возможности моделирования движения робота (как поступательного, так и вращательного), а также его отрисовки на изображении. При этом робот должен всё время находиться в пределах изображения и не должен выезжать за его рамки.

Краткие математические сведения

Положение любой точки A абсолютно твёрдого тела с локальной системой координат с началом в точке O' в рамках сложного плоскопараллельного движения можно представить в векторном виде

$$\vec{r}_A(\tau)\big|_{\tau=t} = \vec{r}_A = \vec{r}_{O'} + \vec{r}_A^{(O')},$$

или в матричном виде в однородных координатах

$$\mathbf{r}_A(\tau) = \begin{pmatrix} x_A(\tau) \\ y_A(\tau) \\ 1 \end{pmatrix}\Bigg|_{\tau=t} = \begin{pmatrix} x_A \\ y_A \\ 1 \end{pmatrix} = H_{O,O'}\,\mathbf{r}_A^{(O')} = \begin{pmatrix} \cos\varphi & -\sin\varphi & x_{O'} \\ \sin\varphi & \cos\varphi & y_{O'} \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x_A^{(O')} \\ y_A^{(O')} \\ 1 \end{pmatrix},$$

где $\vec{r}_{O'}$ — радиус-вектор начала локальной системы координат;

$\vec{r}_A^{(O')}$ — радиус-вектор точки A в локальной системе координат;

$H_{O,O'}$ — матрица перехода из локальной системы координат в абсолютную;

φ — угол поворота локальной системы координат относительно абсолютной.

Таким образом в k-ый момент времени положение любой точки A робота может быть описано по формулам

$$\begin{cases} x_{Ak} = x_{Ck} + x_{Ak}^{(C)}\cos\varphi_k - y_{Ak}^{(C)}\sin\varphi_k \\ y_{Ak} = y_{Ck} + x_{Ak}^{(C)}\sin\varphi_k - y_{Ak}^{(C)}\cos\varphi_k \end{cases},$$

где x_{Ck} и y_{Ck} — координаты центра робота в абсолютной системе координат в k-ый момент времени;

$x_{Ak}^{(C)}$ и $y_{Ak}^{(C)}$ — координаты точки A в системе координат робота в k-ый момент времени;

φ_k — угол поворота робота относительно абсолютной системы координат в k-ый момент времени.

Именно этот принцип будет использоваться при определении положений граничных точек робота и отрисовки его элементов.

Класс Robot

Для моделирования движения и отрисовки робота был разработан класс `Robot` с набором методов, позволяющих осуществлять прямолинейное (`int32_t move(Direction direction)`), вращательное (`int32_t rotate(Rotation rotation)`) и смешанное (`int32_t go(Direction direction, Rotation rotation)`) движение робота, а также методом непосредственно для отрисовки робота на изображении (`int32_t draw(cv::Mat& image)`). Кроме того, класс предоставляет также некоторые другие методы, связанные, в основном, с доступом к его частным членам и с контролем выхода робота за границы изображения.

Заголовочный файл класса `Robot` — `robot.h` — и файл его (класса) реализации `robot.cpp` представлены ниже в листингах 1 и 2.

Листинг 1 — Заголовочный файл класса `Robot`

```
#pragma once

#define _USE_MATH_DEFINES
#include <math.h>

#include "opencv2/core.hpp"
#include "opencv2/highgui.hpp"

#define SPEED 5.0
#define ANGULAR_SPEED 0.1

enum class Direction
{
    FORWARD,
    LEFT,
    BACK,
    RIGHT
};

enum class Rotation
{
    CLOCKWISE,
    COUNTER_CLOCKWISE
};

enum class Quadrant
{
    QUADRANT_I,
    QUADRANT_II,
    QUADRANT_III,
    QUADRANT_IV
};

struct Wheel
{
    float width;
    float diameter;
};
```

```
struct Border
{
    float right;
    float top;
    float left;
    float bottom;
};

class Robot
{
public:
    Robot(
        const float width = 60,
        const float length = 120,
        const Wheel wheel = { 10, 40 },
        const cv::Point2f center = cv::Point2f(0, 0),
        const float angle = M_PI_2,
        const float speed = SPEED,
        const float angularSpeed = ANGULAR_SPEED
    );
    ~Robot() = default;

    void setSpeed(const float speed);
    float speed() const;

    void setAngularSpeed(const float speed);
    float angularSpeed() const;

    void setArea(cv::Size2i area);
    int32_t setArea(cv::Mat image);
    cv::Size2i area() const;

    int32_t setCenter(float centerX, float centerY);
    int32_t setCenter(cv::Mat image);
    cv::Point2f center() const;

    void setBorder(const Border border);
    Border border() const;

    float angle() const;
    float width() const;
    float length() const;
    Wheel wheel() const;

    int32_t move(Direction direction);
    int32_t rotate(Rotation rotation);
    int32_t go(Direction direction, Rotation rotation);
    virtual int32_t draw(cv::Mat& image);
    virtual void doSomething(const char key);
    float calculateDisplacement(Direction direction);
    float calculateAngularDisplacement(Rotation rotation);
    virtual std::vector<cv::Point2f> boundaryPoints();

private:
    cv::Point2f m_center;
    float m_angle;
    const float m_width;
    const float m_length;
    const Wheel m_wheel;
    float m_speed;
    float m_angularSpeed;
    cv::Size2i m_area;
    std::vector<cv::Point2f> m_boundaryPoints;
    Border m_border;
};
```

Листинг 2 — Файл реализации класса Robot

```
#include "robot.h"
#include "opencv2/imgproc.hpp"

#define ZERO 0.000001

using namespace std;
using namespace cv;

Robot::Robot(
    const float width,
    const float length,
    const Wheel wheel,
    const cv::Point2f center,
    const float angle,
    const float speed, const float angularSpeed
) :
    m_width(width),
    m_length(length),
    m_wheel(wheel),
    m_center(center),
    m_angle(angle),
    m_speed(speed),
    m_angularSpeed(speed)
{
    auto white = Scalar(0xFF, 0xFF, 0xFF);
    auto size = Size(1080, 720);
    auto area = Mat(size, CV_8UC3, white);

    setArea(area);
    setCenter(area);

    m_boundaryPoints = boundaryPoints();
}

void Robot::setSpeed(const float speed)
{
    m_speed = speed;
}

float Robot::speed() const
{
    return m_speed;
}

void Robot::setAngularSpeed(const float angularSpeed)
{
    m_angularSpeed = angularSpeed;
}

float Robot::angularSpeed() const
{
    return m_angularSpeed;
}

void Robot::setArea(Size2i area)
{
    m_area = area;
}
```

```
int32_t Robot::setArea(Mat image)
{
    if (image.empty() == true)
    {
        return -1;
    }
    m_area.width = image.cols;
    m_area.height = image.rows;

    Border border =
    {
        static_cast<float>(area().width) - 1.0f,
        static_cast<float>(area().height) - 1.0f,
        0.0,
        0.0
    };
    setBorder(border);

    return 0;
}

Size2i Robot::area() const
{
    return m_area;
}

int32_t Robot::setCenter(float centerX, float centerY)
{
    m_center.x = centerX;
    m_center.y = centerY;

    return 0;
}

int32_t Robot::setCenter(cv::Mat image)
{
    if (image.empty() == true)
    {
        return -1;
    }

    m_center.x = static_cast<float>(image.cols / 2.0);
    m_center.y = static_cast<float>(image.rows / 2.0);

    return 0;
}

Point2f Robot::center() const
{
    return m_center;
}

void Robot::setBorder(const Border border)
{
    m_border = border;
}

Border Robot::border() const
{
    return m_border;
}
```

```
int32_t Robot::move(Direction direction)
{
    float distance = calculateDisplacement(direction);

    switch (direction)
    {
    case Direction::FORWARD:
        m_center.x += distance * cosf(m_angle);
        m_center.y += distance * sinf(m_angle);
        break;
    case Direction::BACK:
        m_center.x -= distance * cosf(m_angle);
        m_center.y -= distance * sinf(m_angle);
        break;
    case Direction::LEFT:
        m_center.x -= distance * sinf(m_angle);
        m_center.y += distance * cosf(m_angle);
        break;
    case Direction::RIGHT:
        m_center.x += distance * sinf(m_angle);
        m_center.y -= distance * cosf(m_angle);
        break;
    default:
        return -1;
    }

    m_boundaryPoints = boundaryPoints();

    if (distance < m_speed)
    {
        return -2;
    }

    return 0;
}

int32_t Robot::rotate(Rotation rotation)
{
    float angle = calculateAngularDisplacement(rotation);

    switch (rotation)
    {
    case Rotation::CLOCKWISE:
        m_angle -= angle;
        break;
    case Rotation::COUNTER_CLOCKWISE:
        m_angle += angle;
        break;
    default:
        return -1;
    }

    m_boundaryPoints = boundaryPoints();

    if (angle < m_angularSpeed)
    {
        return -2;
    }

    return 0;
}
```

```
int32_t Robot::go(Direction direction, Rotation rotation)
{
    move(direction);
    rotate(rotation);

    return 0;
}

int32_t Robot::draw(cv::Mat &image)
{
    if (image.empty() == true)
    {
        return -1;
    }

    if (image.cols != m_area.width || image.rows != m_area.height)
    {
        return -2;
    }

    auto point = [this](const Point2f poligonCenter, const float x, const float y)
    {
        auto point = cv::Point2f();
        point.x = center().x + (x + poligonCenter.x) * cosf(angle()) - (y + poligonCenter.y)
* sinf(angle());
        point.y = center().y + (x + poligonCenter.x) * sinf(angle()) + (y + poligonCenter.y)
* cosf(angle());

        point.y = static_cast<float>(m_area.height) - 1.0f - point.y;
        return point;
    };

    auto poligon = [](cv::Mat& image, vector<Point2f>& poligon, const Scalar& color)
    {
        line(image, poligon.front(), poligon.back(), color);
        for (int32_t index = 1; index < poligon.size(); index++)
        {
            line(image, poligon.at(index - 1), poligon.at(index), color);
        }
    };

    auto black = Scalar(0x00, 0x00, 0x00);

    vector<Point2f> hull =
    {
        point(Point2f(),  m_length / 2.0f,  m_width / 2.0f),
        point(Point2f(), -m_length / 2.0f,  m_width / 2.0f),
        point(Point2f(), -m_length / 2.0f, -m_width / 2.0f),
        point(Point2f(),  m_length / 2.0f, -m_width / 2.0f)
    };

    poligon(image, hull, black);

    vector<Point2f> wheelCenter =
    {
        Point2f( (m_length - m_wheel.diameter) / 2.0f,  (m_width / 2.0f + m_wheel.width)),
        Point2f(-(m_length - m_wheel.diameter) / 2.0f,  (m_width / 2.0f + m_wheel.width)),
        Point2f(-(m_length - m_wheel.diameter) / 2.0f, -(m_width / 2.0f + m_wheel.width)),
        Point2f( (m_length - m_wheel.diameter) / 2.0f, -(m_width / 2.0f + m_wheel.width))
    };
```

```
    for (auto currentWheelCenter : wheelCenter)
    {
        vector<Point2f> currentWheel =
        {
            point(currentWheelCenter, -m_wheel.diameter / 2.0f,  m_wheel.width / 2.0f),
            point(currentWheelCenter, -m_wheel.diameter / 2.0f, -m_wheel.width / 2.0f),
            point(currentWheelCenter,  m_wheel.diameter / 2.0f, -m_wheel.width / 2.0f),
            point(currentWheelCenter,  m_wheel.diameter / 2.0f,  m_wheel.width / 2.0f)
        };

        poligon(image, currentWheel, black);
    }

    return 0;
}

float Robot::angle() const
{
    return m_angle;
}

float Robot::width() const
{
    return m_width;
}

float Robot::length() const
{
    return m_length;
}

Wheel Robot::wheel() const
{
    return m_wheel;
}

float Robot::calculateDisplacement(Direction direction)
{
    auto borderPoint = [this](const float angle)
    {
        auto borderPoint = Point2f();
        if (cosf(angle) >= 0.0 && sinf(angle) >= 0.0)
        {
            borderPoint.x = border().right;
            borderPoint.y = border().top;
        }
        if (cosf(angle) <  0.0 && sinf(angle) >= 0.0)
        {
            borderPoint.x = border().left;
            borderPoint.y = border().top;
        }
        if (cosf(angle) <  0.0 && sinf(angle) <  0.0)
        {
            borderPoint.x = border().left;
            borderPoint.y = border().bottom;
        }
        if (cosf(angle) >= 0.0 && sinf(angle) <  0.0)
        {
            borderPoint.x = border().right;
            borderPoint.y = border().bottom;
        }
        return borderPoint;
    };
```

```
    float distance = m_speed;
    float angle = m_angle + static_cast<uint32_t>(direction) * M_PI_2;

    for (auto& point : m_boundaryPoints)
    {
        float realDistance = FLT_MAX;

        if (fabs(cosf(angle)) > ZERO)
        {
            realDistance = (borderPoint(angle).x - point.x) / cosf(angle);
            if (distance > realDistance)
            {
                distance = realDistance;
            }
        }

        if (fabs(sinf(angle)) > ZERO)
        {
            realDistance = (borderPoint(angle).y - point.y) / sinf(angle);
            if (distance > realDistance)
            {
                distance = realDistance;
            }
        }
    }

    return distance;
}

float Robot::calculateAngularDisplacement(Rotation rotation)
{
    float angle = m_angularSpeed;

    for (auto& point : m_boundaryPoints)
    {
        float radius = hypotf(point.x - m_center.x, point.y - m_center.y);

        auto distance = [this](Quadrant quadrant)
        {
            switch (quadrant)
            {
            case Quadrant::QUADRANT_I:
                return fabs(center().x - border().left);
            case Quadrant::QUADRANT_II:
                return fabs(center().y - border().bottom);
            case Quadrant::QUADRANT_III:
                return fabs(center().x - border().right);
            case Quadrant::QUADRANT_IV:
                return fabs(center().y - border().top);
            default:
                return FLT_MAX;
            }
        };
```

```
        auto realAngle = [this, distance, &radius, &angle](Point2f point,
                                                           Rotation rotation,
                                                           Quadrant quadrant)
        {
            float realAngle = angle;
            if (distance(quadrant) < radius)
            {
                float alpha = static_cast<int32_t>(quadrant) * M_PI_2;
                float phi = atan2f((point.y - m_center.y) * cosf(alpha) -
                                   (point.x - m_center.x) * sinf(alpha)   ,
                                   (point.y - m_center.y) * sinf(alpha) +
                                   (point.x - m_center.x) * cosf(alpha)   );
                float dPhi = acosf(distance(quadrant) / radius);
                realAngle = M_PI - dPhi - (static_cast<int32_t>(rotation) * 2 - 1) * phi;
            }
            if (angle > realAngle)
            {
                return realAngle;
            }
            return angle;
        };

        angle = realAngle(point, rotation, Quadrant::QUADRANT_I  );
        angle = realAngle(point, rotation, Quadrant::QUADRANT_II );
        angle = realAngle(point, rotation, Quadrant::QUADRANT_III);
        angle = realAngle(point, rotation, Quadrant::QUADRANT_IV );
    }

    return angle;
}

vector<Point2f> Robot::boundaryPoints()
{
    auto point = [this](const float x, const float y)
    {
        auto point = cv::Point2f();
        point.x = m_center.x + x * cosf(m_angle) - y * sinf(m_angle);
        point.y = m_center.y + x * sinf(m_angle) + y * cosf(m_angle);

        return point;
    };

    vector<Point2f> points =
    {
        point( m_length / 2.0f,  (m_width + 3.0f * m_wheel.width) / 2.0f),
        point(-m_length / 2.0f,  (m_width + 3.0f * m_wheel.width) / 2.0f),
        point(-m_length / 2.0f, -(m_width + 3.0f * m_wheel.width) / 2.0f),
        point( m_length / 2.0f, -(m_width + 3.0f * m_wheel.width) / 2.0f)
    };

    return points;
}
```

```
void Robot::doSomething(const char key)
{
    switch (key)
    {
    case 'w':
    case 'W':
        move(Direction::FORWARD);
        break;
    case 's':
    case 'S':
        move(Direction::BACK);
        break;
    case 'a':
    case 'A':
        move(Direction::LEFT);
        break;
    case 'd':
    case 'D':
        move(Direction::RIGHT);
        break;
    case 'q':
    case 'Q':
        go(Direction::FORWARD, Rotation::COUNTER_CLOCKWISE);
        break;
    case 'e':
    case 'E':
        go(Direction::FORWARD, Rotation::CLOCKWISE);
        break;
    case 'z':
    case 'Z':
        go(Direction::BACK, Rotation::CLOCKWISE);
        break;
    case 'x':
    case 'X':
        go(Direction::BACK, Rotation::COUNTER_CLOCKWISE);
        break;
    case '.':
    case '>':
        rotate(Rotation::CLOCKWISE);
        break;
    case ',':
    case '<':
        rotate(Rotation::COUNTER_CLOCKWISE);
        break;
    default:
        break;
    }
}
```

Класс WarRobot

Далее был разработан класс `WarRobot` — класс-наследник класса `Robot`, имеющий среди частных членов объект класса `CombatModule`. Класс `CombatModule` предоставляет методы для моделирования движения (вращения) и отрисовки башни с боевым модулем для класса `WarRobot` — боевого робота. Класс `WarRobot` в свою очередь предоставляет те же методы, что и родительский класс `Robot`, а также некоторые другие методы, связанные, например, с доступом к методам объекта класса `CombatModule` или расчёта его положения.

Заголовочный файл и файл реализации класса `CombatModule` `combat_module.h` и `combat_module.cpp` представлены ниже в листингах 3 и 4.

Листинг 3 — Заголовочный файл класса `CombatModule`

```
#pragma once

#include "robot.h"

class CombatModule
{
public:
    CombatModule(
        const float width = 40,
        const float length = 60,
        const cv::Point2f center = cv::Point2f(0, 0),
        const float angle = 0,
        const float angularSpeed = SPEED,
        const Border border = { FLT_MAX, FLT_MAX, -FLT_MAX, -FLT_MAX });
    ~CombatModule() = default;

    int32_t rotate(Rotation rotation);
    void rotateBoundaryPoints(const float angle);

    std::vector<cv::Point2f> towerPoints();
    std::vector<cv::Point2f> gunPoints();

    void setAngularSpeed(const float speed);
    float angularSpeed() const;

    void setCenter(const cv::Point2f center);
    cv::Point2f center() const;

    void setAngle(const float angle);
    float angle() const;

    void setBorder(const Border border);
    Border border() const;

    float width() const;
    float length() const;

    float calculateAngularDisplacement(Rotation rotation);
    std::vector<cv::Point2f> boundaryPoints();
```

```
private:
    cv::Point2f m_center;
    const float m_width;
    const float m_length;
    float m_angle;
    float m_angularSpeed;
    std::vector<cv::Point2f> m_boundaryPoints;
    Border m_border;
};
```

Листинг 4 — Файл реализации класса `CombatModule`

```
#include "combat_module.h"
#include "opencv2/imgproc.hpp"

using namespace cv;
using namespace std;

CombatModule::CombatModule(
    const float width,
    const float length,
    const cv::Point2f center,
    const float angle,
    const float angularSpeed,
    const Border border
) :
    m_width(width),
    m_length(length),
    m_center(center),
    m_angle(angle),
    m_angularSpeed(angularSpeed),
    m_border(border)
{
    m_boundaryPoints = boundaryPoints();
}


void CombatModule::setAngularSpeed(const float angularSpeed)
{
    m_angularSpeed = angularSpeed;
}

float CombatModule::angularSpeed() const
{
    return m_angularSpeed;
}

int32_t CombatModule::rotate(Rotation rotation)
{
    auto rotate = [this](Point2f& point, const float angle)
    {
        auto bufferPoint = point;

        point.x = bufferPoint.x * cosf(angle) - bufferPoint.y * sinf(angle);
        point.y = bufferPoint.x * sinf(angle) + bufferPoint.y * cosf(angle);
    };

    float angle = calculateAngularDisplacement(rotation);
```

```
    switch (rotation)
    {
        case Rotation::CLOCKWISE:
        {
            m_angle -= angle;
            break;
        }
        case Rotation::COUNTER_CLOCKWISE:
        {
            m_angle += angle;
            break;
        }
        default:
        {
            return -1;
        }
    }

    for (auto& point : m_boundaryPoints)
    {
        rotate(point, (static_cast<int32_t>(rotation) * 2 - 1) * angle);
    }

    if (angle < m_angularSpeed)
    {
        return -2;
    }

    return 0;
}

void CombatModule::rotateBoundaryPoints(const float angle)
{
    auto rotate = [this](Point2f& point, const float angle)
    {
        auto bufferPoint = point;

        point.x = bufferPoint.x * cosf(angle) - bufferPoint.y * sinf(angle);
        point.y = bufferPoint.x * sinf(angle) + bufferPoint.y * cosf(angle);
    };

    m_boundaryPoints = boundaryPoints();

    for (auto& point : m_boundaryPoints)
    {
        rotate(point, angle);
    }
}

vector<cv::Point2f> CombatModule::towerPoints()
{
    auto point = [this](const float x, const float y)
    {
        auto point = cv::Point2f();
        point.x = x * cosf(angle()) - y * sinf(angle());
        point.y = x * sinf(angle()) + y * cosf(angle());
        return point;
    };
```

```
    vector<Point2f> points =
    {
        point( m_length / 2.0f,  m_width / 4.0f),
        point( 0.0            ,  m_width / 2.0f),
        point(-m_length / 2.0f,  m_width / 4.0f),
        point(-m_length / 2.0f, -m_width / 4.0f),
        point( 0.0            , -m_width / 2.0f),
        point( m_length / 2.0f, -m_width / 4.0f)
    };

    return points;
}

vector<cv::Point2f> CombatModule::gunPoints()
{
    auto point = [this](const float x, const float y)
    {
        auto point = cv::Point2f();
        point.x = (x + length()) * cosf(angle()) - y * sinf(angle());
        point.y = (x + length()) * sinf(angle()) + y * cosf(angle());
        return point;
    };

    vector<Point2f> points =
    {
        point( m_length / 2.0f,  m_width / 12.0f),
        point(-m_length / 2.0f,  m_width / 12.0f),
        point(-m_length / 2.0f, -m_width / 12.0f),
        point( m_length / 2.0f, -m_width / 12.0f)
    };

    return points;
}

void CombatModule::setCenter(const cv::Point2f center)
{
    m_center = center;
}

cv::Point2f CombatModule::center() const
{
    return m_center;
}

void CombatModule::setAngle(const float angle)
{
    m_angle = angle;
}

float CombatModule::angle() const
{
    return m_angle;
}

void CombatModule::setBorder(const Border border)
{
    m_border = border;
}

Border CombatModule::border() const
{
    return m_border;
}
```

```
float CombatModule::width() const
{
    return m_width;
}

float CombatModule::length() const
{
    return m_length;
}

float CombatModule::calculateAngularDisplacement(Rotation rotation)
{
    float angle = m_angularSpeed;

    for (auto& point : m_boundaryPoints)
    {
        float radius = hypotf(point.x - m_center.x, point.y - m_center.y);

        auto distance = [this](Quadrant quadrant)
        {
            switch (quadrant)
            {
            case Quadrant::QUADRANT_I:
                return fabs(center().x - border().left);
            case Quadrant::QUADRANT_II:
                return fabs(center().y - border().bottom);
            case Quadrant::QUADRANT_III:
                return fabs(center().x - border().right);
            case Quadrant::QUADRANT_IV:
                return fabs(center().y - border().top);
            default:
                return FLT_MAX;
            }
        };

        auto realAngle = [this, distance, &radius, &angle](Point2f point,
                                                           Rotation rotation,
                                                           Quadrant quadrant)
        {
            float realAngle = angle;
            if (distance(quadrant) < radius)
            {
                float alpha = static_cast<int32_t>(quadrant) * M_PI_2;
                float phi = atan2f((point.y - m_center.y) * cosf(alpha) –
                                   (point.x - m_center.x) * sinf(alpha)   ,
                                   (point.y - m_center.y) * sinf(alpha) +
                                   (point.x - m_center.x) * cosf(alpha)   );
                float dPhi = acosf(distance(quadrant) / radius);
                realAngle = M_PI - dPhi - (static_cast<int32_t>(rotation) * 2 - 1) * phi;
            }
            if (angle > realAngle)
            {
                return realAngle;
            }
            return angle;
        };

        angle = realAngle(point, rotation, Quadrant::QUADRANT_I  );
        angle = realAngle(point, rotation, Quadrant::QUADRANT_II );
        angle = realAngle(point, rotation, Quadrant::QUADRANT_III);
        angle = realAngle(point, rotation, Quadrant::QUADRANT_IV );
    }

    return angle;
}
```

```
vector<Point2f> CombatModule::boundaryPoints()
{
    vector<Point2f> points;

    auto tower = towerPoints();
    points.insert(points.end(), tower.begin(), tower.end());

    auto gun = gunPoints();
    points.insert(points.end(), gun.begin(), gun.end());

    return points;
}
```

Заголовочный файл и файл реализации класса `WarRobot` `war_robot.h` и `war_robot.cpp` представлены ниже в листингах 5 и 6.

Листинг 5 — Заголовочный файл класса `WarRobot`

```
#pragma once

#include "robot.h"
#include "combat_module.h"

class WarRobot : public Robot
{
public:
    WarRobot(
        const float width = 60,
        const float length = 120,
        const Wheel wheel = { 10, 40 },
        const CombatModule combatModule = CombatModule(),
        const cv::Point2f center = cv::Point2f(0, 0),
        const float angle = M_PI_2,
        const float speed = SPEED,
        const float angularSpeed = ANGULAR_SPEED
    );
    ~WarRobot() = default;

    CombatModule& combatModule();

    int32_t draw(cv::Mat& image);

    void doSomething(const char key);

    std::vector<cv::Point2f> boundaryPoints();

private:
    CombatModule m_combatModule;
};
```

Листинг 6 — Файл реализации класса `WarRobot`

```
#include "war_robot.h"
#include "opencv2/imgproc.hpp"

using namespace cv;
using namespace std;

WarRobot::WarRobot(
    const float width,
    const float length,
    const Wheel wheel,
    const CombatModule combatModule,
    const cv::Point2f center,
    const float angle,
    const float speed,
    const float angularSpeed
) :
    Robot(width, length, wheel, center, angle, speed, angularSpeed),
    m_combatModule(combatModule)
{

}

CombatModule& WarRobot::combatModule(void)
{
    return m_combatModule;
}

int32_t WarRobot::draw(cv::Mat& image)
{
    if (image.empty() == true)
    {
        return -1;
    }

    if (image.cols != area().width || image.rows != area().height)
    {
        return -2;
    }

    auto point = [this](const Point2f poligonCenter, const float x, const float y)
    {
        auto point = cv::Point2f();
        point.x = center().x + (x + poligonCenter.x) * cosf(angle()) - (y + poligonCenter.y)
* sinf(angle());
        point.y = center().y + (x + poligonCenter.x) * sinf(angle()) + (y + poligonCenter.y)
* cosf(angle());

        point.y = static_cast<float>(area().height) - 1.0f - point.y;
        return point;
    };

    auto poligon = [](cv::Mat& image, vector<Point2f>& poligon, const Scalar& color)
    {
        line(image, poligon.front(), poligon.back(), color);
        for (int32_t index = 1; index < poligon.size(); index++)
        {
            line(image, poligon.at(index - 1), poligon.at(index), color);
        }
    };

    auto black = Scalar(0x00, 0x00, 0x00);
```

```
    vector<Point2f> hull =
    {
        point(Point2f(),  length() / 2.0f,  width() / 2.0f),
        point(Point2f(), -length() / 2.0f,  width() / 2.0f),
        point(Point2f(), -length() / 2.0f, -width() / 2.0f),
        point(Point2f(),  length() / 2.0f, -width() / 2.0f)
    };

    poligon(image, hull, black);

    vector<Point2f> wheelCenter =
    {
        Point2f( (length() - wheel().diameter) / 2.0f,  (width() / 2.0f + wheel().width)),
        Point2f(-(length() - wheel().diameter) / 2.0f,  (width() / 2.0f + wheel().width)),
        Point2f(-(length() - wheel().diameter) / 2.0f, -(width() / 2.0f + wheel().width)),
        Point2f( (length() - wheel().diameter) / 2.0f, -(width() / 2.0f + wheel().width))
    };

    for (auto currentWheelCenter : wheelCenter)
    {
        vector<Point2f> currentWheel =
        {
            point(currentWheelCenter, -wheel().diameter / 2.0f,  wheel().width / 2.0f),
            point(currentWheelCenter, -wheel().diameter / 2.0f, -wheel().width / 2.0f),
            point(currentWheelCenter,  wheel().diameter / 2.0f, -wheel().width / 2.0f),
            point(currentWheelCenter,  wheel().diameter / 2.0f,  wheel().width / 2.0f)
        };

        poligon(image, currentWheel, black);
    }

    auto towerPoints = m_combatModule.towerPoints();

    vector<Point2f> tower;
    for (auto currentPoint : towerPoints)
    {
        tower.push_back(point(combatModule().center(), currentPoint.x, currentPoint.y));
    }

    poligon(image, tower, black);

    auto gunPoints = m_combatModule.gunPoints();

    vector<Point2f> gun;
    for (auto currentPoint : gunPoints)
    {
        gun.push_back(point(combatModule().center(), currentPoint.x, currentPoint.y));
    }

    poligon(image, gun, black);

    return 0;
}

vector<Point2f> WarRobot::boundaryPoints()
{
    auto point = [this](const float x, const float y)
    {
        auto point = cv::Point2f();
        point.x = center().x + x * cosf(angle()) - y * sinf(angle());
        point.y = center().y + x * sinf(angle()) + y * cosf(angle());

        return point;
    };
```

```
    vector<Point2f> points =
    {
        point( length() / 2.0f,  (width() + 3.0f * wheel().width) / 2.0f),
        point(-length() / 2.0f,  (width() + 3.0f * wheel().width) / 2.0f),
        point(-length() / 2.0f, -(width() + 3.0f * wheel().width) / 2.0f),
        point( length() / 2.0f, -(width() + 3.0f * wheel().width) / 2.0f),
    };

    auto gunPoints = m_combatModule.gunPoints();
    for (auto& currentPoint : gunPoints)
    {
        currentPoint = point(combatModule().center().x + currentPoint.x,
                             combatModule().center().y + currentPoint.y);
    }
    points.insert(points.end(), gunPoints.begin(), gunPoints.end());

    auto towerPoint = point(combatModule().center().x, combatModule().center().y);
    Border border =
    {
        -(towerPoint.y - this->border().top   ),
         (towerPoint.x - this->border().left  ),
        -(towerPoint.y - this->border().bottom),
         (towerPoint.x - this->border().right )
    };
    m_combatModule.setBorder(border);
    m_combatModule.rotateBoundaryPoints(angle() - M_PI_2);

    return points;
}

void WarRobot::doSomething(const char key)
{
    switch (key)
    {
    case 'w':
    case 'W':
        move(Direction::FORWARD);
        break;
    case 's':
    case 'S':
        move(Direction::BACK);
        break;
    case 'a':
    case 'A':
        move(Direction::LEFT);
        break;
    case 'd':
    case 'D':
        move(Direction::RIGHT);
        break;
    case 'q':
    case 'Q':
        go(Direction::FORWARD, Rotation::COUNTER_CLOCKWISE);
        break;
    case 'e':
    case 'E':
        go(Direction::FORWARD, Rotation::CLOCKWISE);
        break;
    case 'z':
    case 'Z':
        go(Direction::BACK, Rotation::CLOCKWISE);
        break;
```

```
    case 'x':
    case 'X':
        go(Direction::BACK, Rotation::COUNTER_CLOCKWISE);
        break;
    case '.':
    case '>':
        rotate(Rotation::CLOCKWISE);
        break;
    case ',':
    case '<':
        rotate(Rotation::COUNTER_CLOCKWISE);
        break;
    case ']':
    case '}':
    {
        combatModule().rotate(Rotation::CLOCKWISE);
        break;
    }
    case '[':
    case '{':
        combatModule().rotate(Rotation::COUNTER_CLOCKWISE);
        break;
    default:
        break;
    }
}
```