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

ОТЧЁТ ПО НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

Разработка виртуальной модели системы кругового обзора мобильного робота

Выполнил
студент гр.3331506/60401 Г.А. Куршаков

Руководитель
старший преподаватель А.С. Габриель

Научный консультант В.В, Варлашин

«___» _____ 201__ г.

Санкт-Петербург
2019

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

ЗАДАНИЕ
на выполнение научно-исследовательской работы
Куршакову Георгию Алекснадровичу
студенту гр. 3331506/60401

1 Тема работы
Разработка виртуальной модели системы кругового обзора мобильного робота

2 Срок сдачи студентом законченной работы
12.2019

3 Исходные данные к работе
Требования к виртуальной модели, внешнее устройство мобильного робота.

4 Содержание расчетно-пояснительной записки
Обзор мирового опыта применения оптических систем мобильных роботов и их моделирования, математических моделей камер и основных преобразований, выполняемых над изображениями; реализация виртуальной модели системы кругового обзора; исследование влияния размеров входных изображений на частоту кадров результирующего изображения.

5 Перечень графического материала (с точным указанием обязательных чертежей)
нет

6 Консультанты по работе
Варлашин Виктор Витальевич, ЦНИИ РТК, младший научный сотрудник

7 Дата выдачи задания 09.09.19

Руководитель Габриель Антон Сергеевич, старший преподаватель
(ФИО, должность, подпись руководителя)

Задание принял к исполнению 09.09.19
(дата)

(подпись студента)

РЕФЕРАТ

58 с., 22 рис., 5 табл., 21 источник

ВИЗУАЛИЗАЦИЯ, ВИРТУАЛЬНАЯ РЕАЛЬНОСТЬ, СКЛЕЙКА ИЗОБРАЖЕНИЙ, ПАНОРАМА, СИСТЕМА КРУГОВОГО ОБЗОРА, НАВИГАЦИЯ.

В работе рассматривается создание виртуальной модели системы кругового обзора мобильного робота. Виртуальная модель призвана упростить процесс разработки, тестирования и отладки такой системы применительно к мобильному робототехническому комплексу. В рамках работы рассмотрены современные телевизионные системы мобильных роботов, варианты расположения камер системы кругового обзора, выбрана среда разработки, разработана виртуальная модель мобильного робота с расположенными на нём камерами, реализован способ передачи изображений с виртуальных камер для обработки алгоритмами компьютерного зрения. Проведено измерение времени работы алгоритма передачи изображения для обработки. Также рассмотрены основные преобразования двумерных изображений и способы построения панорам.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Значительная часть современных мобильных роботов является телеуправляемыми, то есть управляется оператором дистанционно, часто без возможности оценить обстановку со стороны. Для оценки окружающей обстановки используются различные информационно-измерительные устройства и системы.

Наибольшее распространение получают оптические системы различной конфигурации, из которых наиболее полную информацию оператору предоставляют системы кругового обзора, позволяющие охватить всё окружающее пространство по горизонтали. Такие системы, несмотря на их преимущества, ещё не дают оператору возможности изменить ракурс или угол обзора, что может быть важно при ответственных манёврах и манипуляциях. Отработку алгоритма, позволяющего оператору наблюдать за роботом от третьего лица с возможностью изменения угла обзора, имеет смысл производить на виртуальной модели.

Виртуальная реальность активно применяется в современном мире для выполнения различных задач: например, для тренировки военных и сотрудников органов правопорядка, в промышленности при разработке, моделировании и прототипировании, в Интернет-торговле, где у покупателя есть возможность «примерить» выбранные предметы одежды и, разумеется, в сфере образования (от виртуальных путешествий по солнечной системе до подробных моделей человеческого тела для студентов медицинских направлений) [1]. В последние годы получает особое распространение виртуальное моделирование технических систем и виртуальные эксперименты.

Особенность и основное преимущество виртуального эксперимента перед реальным заключается в высокой воспроизводимости и повторяемости их результатов, поскольку виртуальная экспериментальная установка позволяет полностью исключить влияние на эксперимент нежелательных

внешних факторов. Кроме того, виртуальные эксперименты, как правило, способны значительно экономить материальные и временные затраты [2].

Виртуальная модель, имитирующая робот и его окружение и позволяющая передавать изображения с виртуальных камер для обработки алгоритмами технического зрения, способна дать следующие преимущества:

- повторяемость результатов экспериментов (от эксперимента к эксперименту не меняются: траектория движения робота, взаимное расположение объектов, условия виртуальной окружающей среды);

- перенос разработанных алгоритмов с виртуальной модели на реального робота;

- отсутствие необходимости в физическом наличии робота.

Целью работы является реализация виртуальных моделей мобильного робота и окружающей среды с возможностью установки и настройки виртуальных видеокамер и передачи изображений с них для обработки алгоритмами компьютерного зрения.

В ходе работы решаются следующие задачи:

1. Обзор современного состояния телевизионных систем мобильных роботов и алгоритмов, применяемых для обработки изображений в системах кругового обзора;

2. Обоснование выбора программного обеспечения для разработки виртуальной модели;

3. Реализация трёхмерной модели робота с телекамерами и его окружения в рабочей среде;

4. Реализация передачи изображений с виртуальных камер для обработки алгоритмами технического зрения;

5. Исследование влияния алгоритмической обработки изображений на частоту кадров.

1 Обзор мирового опыта применения оптических систем мобильных роботов и их моделирования

1.1 Телевизионные системы мобильных роботов

Управление роботами осуществляется исходя из некоторой информации об окружающем пространстве, получаемой посредством информационно-измерительных устройств. При автоматической навигации и управлении высокую эффективность показывают лидары и ультразвуковые датчики, позволяющие составить представление о расстояниях до окружающих робот объектов. Однако, информация, представленная в таком виде, как правило, менее понятна оператору-человеку. В ответственных ситуациях, например, при радиоактивной разведке или взрывотехнических работах ошибка может привести к неприемлемым последствиям. Наиболее удобной для оператора в большинстве случаев является визуальная форма представления информации, полученная от оптических телевизионных систем.

Изображение, получаемое с помощью видеокамер, установленных на борту мобильного робота, является основным источником информации для оператора. Число и расположение видеокамер может быть различным. Бортовые телевизионные системы классифицируются по различным критериям:

- по количеству телекамер (с одной телекамерой, с несколькими телекамерами);

- по функциональному назначению:

1) одиночные камеры;
2) PTZ-камеры (англ. Pan-Tilt-Zoom);
3) стереокамеры;
4) системы кругового обзора;
5) комбинированные системы [3].

Самым распространенным вариантом является расположение на корпусе робота или манипуляционной системе одиночной телекамеры, изображение с которой транслируется на пульт дистанционного управления. Также встречается объединение нескольких изображений с различных телекамер в одно и его трансляция на ПДУ оператора.

Изображение с телекамер, расположенных на корпусе, часто не позволяет оператору оценить расположение робота относительно объектов окружающей среды. Одним из путей решения этой проблемы является закрепление телекамеры на манипуляторе или выносной штанге. При таком расположении корпус робота оказывается в кадре.

PTZ-камеры дают оператору больше свободы, позволяя менять как угол обзора, так и фокусное расстояние камеры, фокусируясь на важных объектах. Однако, это по-прежнему не даёт возможности охватывать всё окружающее пространство единовременно.

Стереокамеры могут применяться для транслирования стереоизображения на ПДУ или для построения карты глубины.

Системы кругового обзора предназначены для охвата всего окружающего пространства на 360° по горизонтали и до 180° по вертикали. Это направление является новым в робототехнике и пришло в неё из автомобилестроения, где такие системы призваны решить проблемы «слепых» зон и облегчить маневрирование на сложных участках. В частности, такие системы могут использоваться для облегчения управления автомобилем, например, во время парковки [4, 5]. Автомобильные системы кругового обзора, как правило, имеют камеры, расположенные непосредственно на корпусе автомобиля: спереди, сзади и по бокам (рисунок 1) [6]. Разработаны также системы кругового обзора гуманоидных роботов [7].

Комбинированные телевизионные системы также получают распространение. Так, в ходе работы [8] была разработана система кругового

обзора, включающая в себя три стандартных телекамеры и одну стереокамеру для получения более детальной информации.

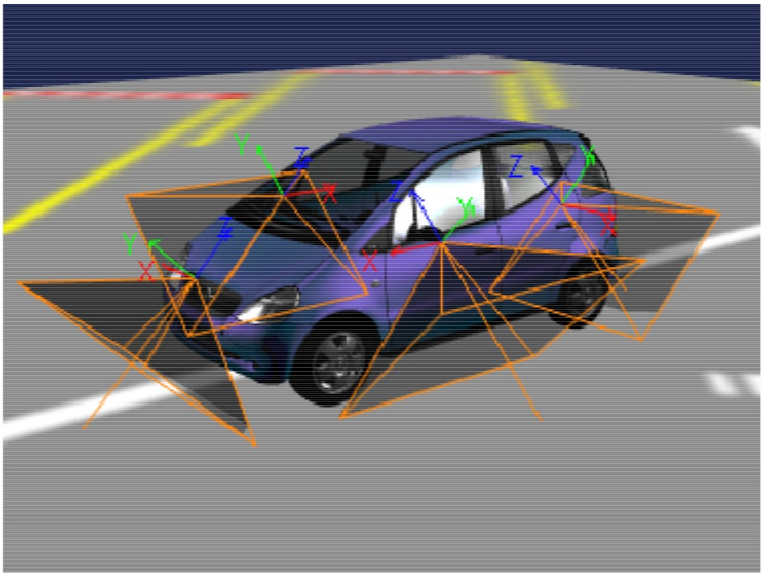

Рисунок 1 – Расположение камер автомобильной системы кругового обзора

Для мобильного робота возможен также вариант расположения камер на выносной штанге. Это наиболее распространённый в настоящее время способ предоставить оператору вид от третьего лица.

Рассмотрим наиболее распространенные на современных МРК телевизионные системы. Согласно статье [9], мобильные роботы подразделяются на классы и подклассы, перечисленные в таблице 1:

Таблица 1 – Классификация мобильных роботов

Класс, подкласс	Общая масса, кг
Сверхлёгкий первый (UL1)	До 1 кг
Сверхлёгкий второй (UL2)	От 1 до 5 кг
Сверхлёгкий третий (UL3)	От 5 до 15 кг
Сверхлёгкий четвёртый (UL4)	От 15 до 50 кг
Сверхлёгкий пятый (UL5)	От 50 до 100 кг
Лёгкий первый (L1)	От 100 до 300 кг
Лёгкий второй (L2)	От 300 до 1000 кг
Средний первый (M1)	От 1000 до 5000 кг
Средний второй (M2)	От 5000 до 20000 кг
Тяжёлый (H)	Более 20000 кг

В этой статье приводятся характеристики некоторых мобильных роботов российского и зарубежного производства. Рассмотрим роботы

сверхлёгкого класса. Наиболее распространенной конфигурацией камер среди них является расположение на корпусе одной курсовой камеры или двух камер: курсовой и кормовой. Встречается вариант с подвижной камерой: камеры сверхлегких роботов «Iris» [10] израильского производства и «Скарабей» [11] российского производства способны поворачиваться вокруг горизонтальной оси.

Роботы Nerva [12] французского производства одни из немногих обладают четырьмя камерами, установленными на корпусе: спереди, сзади и по бокам. Курсовая камера обладает высоким разрешением (1280x720) и подсветкой. Эта система позволяет охватывать 360° по горизонтали и транслировать на ПДУ панорамное изображение.

В качестве примера для построения виртуальной модели взят малогабаритный робототехнический комплекс (МРК) «Капитан» производства ЦНИИ РТК. Он представляет из себя универсальную роботизированную платформу, оснащенную двумя видеокамерами: курсовой и кормовой. Такое расположение камер не даёт возможности достичь охвата 360° по горизонтали. У данного робота существует несколько конфигураций:

1. Манипулирование объектами;
2. Разведка и наблюдение;
3. Инженерная разведка;
4. Взрывотехнические работы.

Вторая конфигурация (разведка и наблюдение) включает в себя трансфокаторную и широкоугольную камеры, значительно увеличивая угол обзора и позволяя оператору его изменять. Однако этот модуль устанавливается на манипуляторе вместо захватного устройства, что делает невозможным одновременное манипулирование объектами [13].

Установка на МРК «Капитан» системы кругового обзора позволила бы значительно упростить дистанционное управление роботом, манипулирование объектами и взаимодействие с окружающей средой.

1.2 Виртуальное моделирование систем кругового обзора

Разработку системы кругового обзора применительно к данному мобильному роботу наиболее целесообразно производить с использованием виртуальной модели. Виртуальное моделирование систем кругового обзора уже получило некоторое распространение в автомобильной промышленности: отработка алгоритмов на виртуальных моделях значительно упрощает разработку, снижает расходы, сопряжённые с использованием реального автомобиля и минимизирует риски для жизни и здоровья людей.

В частности, в работе [6] рассматривается система моделирования монитора кругового обзора, то есть отображения на экране положения транспортного средства вместе с окружающими объектами во время сложных манёвров (вид сверху). Особенностью системы, представленной в этой работе, является её применимость к различным видам транспортных средств, что даёт возможность применить её в том числе и к мобильному роботу.

Разработка виртуальной модели системы кругового обзора требует реализации следующих составных частей:

1. Виртуальная модель мобильного робота;
2. Виртуальные модели камер;
3. Виртуальные модели окружающих объектов;
4. Перемещение модели робота в виртуальном пространстве;
5. Передача изображений для обработки алгоритмами компьютерного зрения.

1.3 Выбор среды разработки

Выбор среды разработки должен быть основан на следующих критериях:

1. Возможность создания или импорта трёхмерных моделей – это условие необходимо для реализации моделей самого мобильного робота и окружающих его объектов;

2. Возможность создания и настройки виртуальных камер – камеры в реальном мире могут иметь различные параметры и особенности расположения;

3. Возможность подключения сторонних библиотек для обработки изображений – изображения должны подвергаться обработке алгоритмами технического зрения перед выводом на дисплей.

4. Моделирование движения манипулятора – учитывая, что модель создается с расчётом на применение к мобильным роботам, следует помнить, что одной из выполняемых роботом задач может быть манипулирование объектами, и при перемещении объектов оператор должен получать наиболее полную информацию о текущем положении манипулятора.

5. Стоимость – в данной работе может быть использовано только бесплатное и свободно распространяемое программное обеспечение.

При выборе рассмотрено несколько категорий программного обеспечения.

В качестве первой категории выберем системы автоматизированного проектирования (САПР), такие как SolidWorks и CATIA. Общими характеристиками для САПР являются широкие возможности для создания трёхмерных моделей. Как SolidWorks, так и CATIA предлагают множество инструментов и методов для этого.

SolidWorks предоставляет возможность создания и размещения виртуальных камер, однако доступные для регулирования параметры не включают в себя дисторсию и разрешение, которые являются важными при обработке поступающего изображения. То же самое можно сказать и о CATIA. В целом, САПР в большей степени предназначены для проектирования, чем для виртуального моделирования, и детальная настройка камер не входит в число их возможностей.

Также рассматриваемые САПР не взаимодействуют со сторонними подключаемыми библиотеками, что исключает возможность автоматической обработки изображений. С другой стороны, они поддерживают моделирование движения манипулятора, что является серьёзным преимуществом.

SolidWorks является платной программой, однако предоставляет пробные версии для студентов и начинающих предпринимателей. CATIA также предлагает студенческую версию, которая является платной. В данной работе мы не можем позволить себе использовать платные и пробные версии программ, поскольку нас интересует долговременный результат и все возможности САПР.

Рассмотрим симуляторы для исследований в области автономного вождения: CARLA и NVIDIA Drive Constellation. Будучи ориентированными на автомобильную промышленность, они созданы специально для отработки в том числе систем кругового обзора.

CARLA содержит инструменты для создания моделей транспортных средств, однако они достаточно примитивны и следуют одному паттерну. Создание или импорт трёхмерной модели конкретного мобильного робота требует неоправданно серьёзных сложностей в настройке. Аналогичная ситуация наблюдается у NVIDIA Drive Constellation, но его инструменты ещё более примитивны. CARLA предоставляет широкие возможности для настройки виртуальных камер, которые включают в себя детальные параметры как матрицы камеры, так и линзы. NVIDIA Drive Constellation не даёт таких возможностей. CARLA также, в отличие от NVIDIA Drive Constellation, поддерживает использование сторонних библиотек. К тому же, CARLA является open-source проектом и распространяется свободно. NVIDIA Drive Constellation предоставляется по непосредственному запросу в компанию через форму на сайте.

Недостатком обоих симуляторов является невозможность моделирования движений манипулятора: физика ПО не предусматривает

наличия большого числа подвижных частей на корпусе транспортного средства.

Рассмотрим игровые движки Unity и Unreal Engine. Оба позволяют импортировать трёхмерные модели в расширении FBX. Оба имеют возможность точной настройки камер, при этом Unity обладает функцией «Physical camera», позволяющей настроить размер матрицы и фокусное расстояние. Кроме того, в Unity присутствует возможность постобработки изображения для придания ему дополнительных эффектов, таких как дисторсия. Также оба движка позволяют подключать DLL (динамически подключаемые библиотеки). Физические модели и Unity, и Unreal Engine позволяют моделировать механическое взаимодействие твёрдых тел, включая движение манипулятора. Оба движка являются бесплатными и свободно распространяемыми для некоммерческих целей.

Таким образом, всем вышеперечисленным требованиям удовлетворяют среды разработки видеоигр, и, в частности, среда Unity. В отличие от Unreal Engine, она предоставляет больше возможностей для настройки виртуальных камер. В силу этих причин выбор остановлен на ней.

1.4 Выбор библиотеки алгоритмов технического зрения

Существует несколько наиболее распространённых библиотек, реализующих алгоритмы технического зрения. Среди них можно отметить такие библиотеки как OpenCV, OpenVX, libXCam. OpenCV является наиболее распространённой в этой сфере библиотекой. OpenVX может использоваться как самостоятельная библиотека или как ускоритель для OpenCV на встроенных системах. libXCam предлагает множество разнообразных функций и алгоритмов, однако поддерживается только операционными системами Linux. Все они распространяются бесплатно.

Для разработки программной части виртуальной модели была выбрана библиотека OpenCV. В настоящее время она является самой

распространённой библиотекой, реализующей алгоритмы технического зрения. Также она обладает достаточным быстродействием для обработки изображений в реальном времени и активно поддерживается сообществом как разработчиков, так и многочисленных пользователей.

OpenCV реализована на языках C/C++ и разрабатывается для Python, Java, Ruby, Matlab, Lua и других языков. В данной работе используется язык C++. Он позволяет получить большее быстродействие функций и алгоритмов, чем любой другой язык, и создать библиотеки динамической компоновки, которые будут подключены к среде разработки.

Для написания программного кода и его отладки используется интегрированная среда разработки Visual Studio 2017. Её преимуществами являются возможность подключения для отладки к сторонним процессам, в том числе к Unity, и дополнение Image Watch 2017, которое даёт возможность просмотреть графическое представление содержимого массива типа cv::Mat в режиме отладки.

1.5 Выводы по главе

Аналитический обзор современных телевизионных систем мобильных роботов позволил сделать выводы об их достоинствах и недостатках, а также сферах их применения. Для разработки алгоритма, позволяющего изменять угол обзора выходного изображения системы кругового обзора принято решение прибегнуть к виртуальному моделированию. С целью рассмотрения существующих решений проведен обзор мирового опыта виртуального моделирования систем кругового обзора.

Для моделирования выбрана среда разработки Unity. Для обработки изображений с виртуальных камер выбрана библиотека OpenCV, реализованная на языке C++. Для написания программного кода выбрана среда разработки Visual Studio 2017.

2 Обзор математических моделей камер и основных преобразований, выполняемых над изображениями

2.1 Математическая модель камеры

Традиционные системы визуализации трёхмерных сцен используют точечные камеры, отображающие сцену в соответствии с законами перспективной проекции, и заведомо имеющими угол обзора меньше 180 градусов. В этой модели предполагается, что от каждой точки пространства в отверстие камеры, расположенное на плоскости отверстия попадает лишь один луч, который проецируется на плоскость изображения. Это изображение оказывается перевёрнутым относительно оптической оси камеры. Такой модели по умолчанию соответствуют камеры среды Unity. Точечная модель камеры представлена на рисунке 2 [14], где

f – фокусное расстояние камеры;

Z – расстояние до объекта;

X – линейный размер объекта;

x – линейный размер изображения объекта.

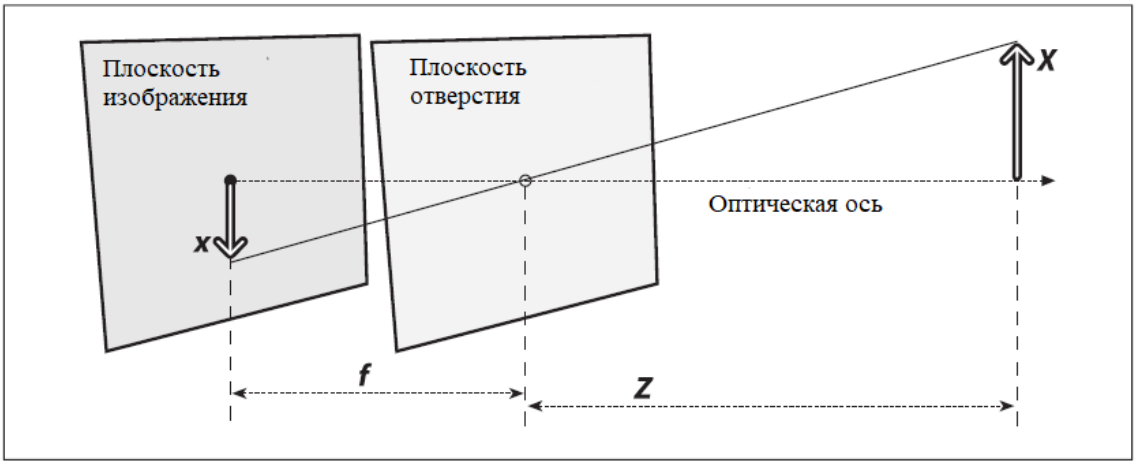

Рисунок 2 – Точечная модель камеры: отверстие пропускает только один луч света от каждой определённой точки в пространстве; эти лучи формируют изображение на плоскости изображения

Любая реальная камера обладает линзой, основная задача которой – фокусировка лучей. Идеальной формой линзы, не дающей искажений, является параболическая, однако такие линзы достаточно сложны в

изготовлении, поэтому гораздо чаще используются линзы сферической формы.

Результатом применения таких линз является дисторсия – искажение изображения в оптической системе. Такое искажение усиливается при удалении от начала координат, и бывает двух типов: отрицательная (подушкообразная) и положительная (бочкообразная). На рисунке 3 представлены искажения исходного изображения при положительной и отрицательной дисторсии.

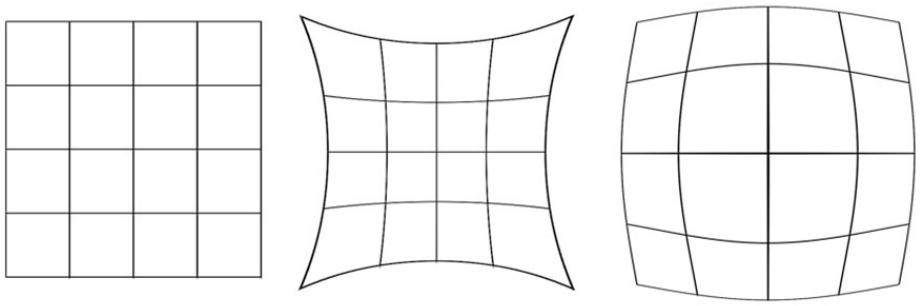

Рисунок 3 – слева направо: исходное изображение квадрата, изображение, полученное с помощью системы с отрицательной дисторсией, изображение, полученное с помощью системы с положительной дисторсией.

Дисторсия характеризуется набором коэффициентов, зная которые, можно вычислить истинные положения точек изображения, соответствующие их положению в реальном мире.

Дисторсия камеры может быть устранена программно, при помощи различных алгоритмов. Этот процесс сопряжён с калибровкой камеры. Для калибровки используется некий заранее известный паттерн, содержащий набор точек, которые могут быть найдены при помощи алгоритмов компьютерного зрения. Чаще всего используется так называемая «шахматная доска», состоящая из чёрных и белых клеток квадратной формы [15].

В реальных роботах, как правило, используются камеры со сверхширокоугольными объективами типа «рыбий глаз». Они имеют угол обзора, превышающий 180 градусов, однако обладают сильной отрицательной дисторсией [16]. В данной научно-исследовательской работе предполагается, что изображение попадает на обработку с уже устранённой дисторсией, однако в дальнейшем алгоритм её устранения может быть включён в программный код.

В библиотеке OpenCV используется следующая модель камеры типа «рыбий глаз». Пусть P – точка в глобальной системе координат, которые хранятся в векторе X. Вектор координат точки P в системе координат камеры выражается следующим образом:

$$X_c = RX + T,$$

где R – матрица поворота, а x, y и z – составляющие вектора X_c:

$$x = X_{c1},$$

$$y = X_{c2},$$

$$z = X_{c3}.$$

Координаты a и b проекции точки P на плоскость изображения описываются следующим образом [17]:

$$a = x / z,$$

$$b = y / z,$$

$$r^2 = a^2 + b^2,$$

$$\theta = arctg(r).$$

Далее рассчитывается дисторсия линзы:

$$\theta_d = \theta\left(1 + k_1\theta^2 + k_2\theta^4 + k_3\theta^6 + k_4\theta^8\right).$$

Координаты точки изображения после дисторсии x' и y':

$$x' = (\theta_d / r)a,$$

$$y' = (\theta_d / r)b.$$

Затем происходит преобразование в пиксельные координаты:

$$u = f_x\left(x' + \alpha y'\right) + c_x,$$

$$v = f_y y' + c_y.$$

Таким образом, видно, что дисторсия в модели, используемой OpenCV, выражается четырьмя параметрами k_1, k_2, k_3, k_4.

В Unity дисторсия может симулироваться с помощью инструментов постобработки. Отличие реализации дисторсии камеры от математической модели OpenCV заключается в том, что она описывается только одним параметром k_1. Добиться соответствия двух моделей предположительно можно, модифицировав исходный программный код постобработки Unity. Окно настройки дисторсии Unity при помощи постобработки представлено на рисунке 4.

Рисунок 4 – Окно настройки дисторсии в Unity

Для настройки дисторсии предлагаются следующие параметры:

- Intensity – степень дисторсии, может быть как положительной, так и отрицательной;

- X Multiplier – множитель по оси X;

- Y Multiplier – множитель по оси Y;

- Center X – смещение центра линзы относительно камеры по оси X;

- Center Y – смещение центра линзы относительно камеры по оси Y;

- Scale – масштаб изображения.

2.2 Преобразования и методы объединения изображений

Существует ряд преобразований, применяемых к изображениям: это, во-первых, перспективные преобразования (гомография), а во-вторых,

трёхмерные преобразования (например, цилиндрическое или сферическое проецирование) [18].

В общем виде перспективные преобразования определяются следующей формулой:

$$[a\,b\,c\,d\,e\,f\,g\,h\,1][x\,y\,1]=[u\,v\,w]$$

Матрица 3x3 называется гомографией, именно её параметры определяют, как именно выглядит применяемое преобразование. x и y – гомогенные координаты пикселя на первоначальном изображении. Координаты пикселя на результирующем изображении определяются как

$$x'=u/w,\ y'=v/w.$$

Перспективное преобразование имеет следующие подвиды:

- смещение (translation): изображение поступательно перемещается на плоскости.

- поворот (rotation): изображение перемещается с поворотом, но без изменения размеров.

- подобие (similarity): изображение перемещается с изменением абсолютных размеров, оставаясь подобным оригиналу.

- аффинное (affine): параллельные линии изображения остаются параллельными (прямоугольник преобразуется в параллелограмм).

- перспективное или гомографическое (perspective) в общем виде: прямые линии остаются прямыми (прямоугольник преобразуется в трапецию).

Результирующее изображение можно получить путём операций над матрицами: имея исходное изображение $f(x)$ и преобразование $x'=h(x)$, необходимо найти трансформированное изображение $g(x')=f(h(x))$.

Система кругового обзора подразумевает другой вид преобразований: наблюдатель (камера) не перемещается поступательно, но вращается вокруг своей вертикальной оси. Для правильной склейки изображений такой панорамы необходимо знать две переменные: фокусное расстояние и расстояние, на которое следует совместить изображения для получения

панорамы. При построении панорам кругового обзора используется два основных метода: цилиндрическая и сферическая панорама [19].

Для построения цилиндрической панорамы каждое изображение проецируется на цилиндрическую поверхность. Затем их стыки объединяются и получается результирующая мозаика (см. рисунок 5) [20].

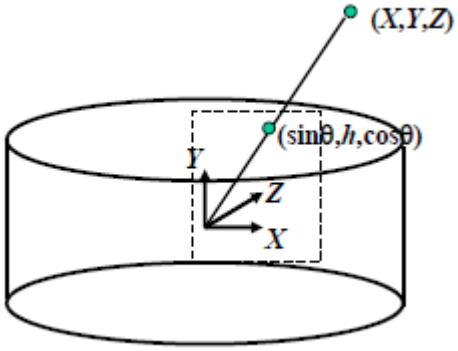

Рисунок 5 – Цилиндрическое преобразование

Математически проецирование описывается следующим образом:

– Точка пространства отображается на цилиндр:

$$(\hat{x}, \hat{y}, \hat{z}) = \frac{1}{\sqrt{\square}}$$

– Переход к цилиндрической системе координат:

$$(\sin\sin\theta, h, \cos\cos\theta) = (\hat{x}, \hat{y}, \hat{z})$$

– Переход к системе координат цилиндрического изображения:

$$(\tilde{x}, \tilde{y}) = (s\theta, sh) + (\tilde{x}_c, \tilde{y}_c)$$

Цилиндрическое преобразование выглядит следующим образом при заданном фокусном расстоянии f и центре изображения (x_c, y_c):

$$\theta = \frac{(x_{cyl} - x_c)}{f}$$

$$h = \frac{(y_{cyl} - y_c)}{f}$$

$$\hat{x} = \sin\sin\theta$$

$$\hat{y} = h$$

$$\hat{z} = \cos\cos\theta$$

$$x = \frac{f\hat{x}}{\hat{z}} + x_c$$

$$y = \frac{f\,\hat{y}}{\hat{z}} + y_c$$

Аналогично строится и сферическая панорама, но изображение проецируется на сферическую поверхность (см. рисунок 6) [21]:

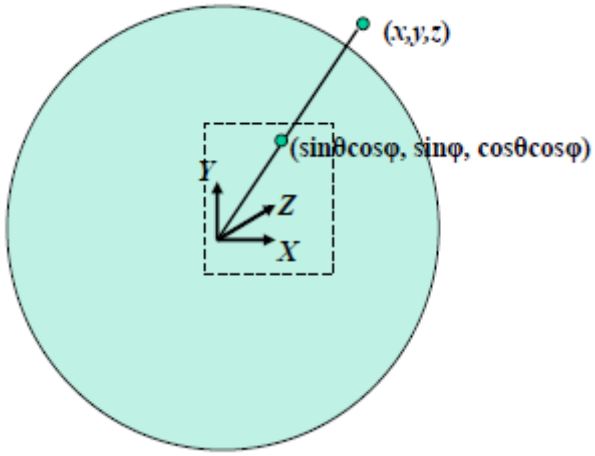

Рисунок 6 – Сферическое преобразование

Первым вариантом данной операции является прямое отображение (forward warping), заключающееся в том, чтобы направить каждый пиксель исходного изображения в соответствующую точку результирующего. Однако у этого варианта есть существенный недостаток: точка на результирующем изображении может не соответствовать конкретному пикселю. У этого недостатка есть решение: учесть вклад значения данного пикселя в значения ближайших к нему, и затем нормализовать вклады, не оставив таким образом пикселей с неопределёнными значениями.

Однако существует и другой вариант: обратное отображение (inverse warping). В таком случае необходимо для каждого пикселя выходного изображения найти соответствующий пиксель исходного. Разумеется, в таком случае тоже есть вероятность попадания «между» пикселей исходного изображения. В качестве нужного значения берётся значение из предварительно интерполированного входного изображения. Существуют различные варианты интерполяционных фильтров, например, билинейный, бикубический. Самым простым вариантом является простое присваивание значение ближайшего соседнего пикселя.

Для корректной склейки, как правило, необходимо найти общие точки на изображениях. После склейки панорамы для получения вида от третьего лица необходимо спроецировать её на некоторую поверхность, которая затем совмещается с трёхмерной моделью робота. Это даст оператору возможность самостоятельно выбирать наиболее удобный угол обзора.

2.3 Выводы по главе

Рассмотрены две математических модели камеры: точечная (pinhole model, модель камеры-обскуры) и камеры с дисторсией. Исследованы отличия в математических моделях дисторсии камеры библиотеки OpenCV и среды Unity.

Также рассмотрены основные виды преобразований над изображениями, в том числе способы их объединения в панорамное изображение.

3 Реализация виртуальной модели системы кругового обзора

3.1 Создание модели мобильного робота

Модель робота не должна быть излишне детализированной и содержать множество элементов, поскольку расчёт их кинематики будет замедлять работу виртуальной модели и при этом не будет нести какой-либо значительной информации. Следует ограничиться только самыми важными подвижными деталями. В рассматриваемом случае это корпус робота, его гусеницы и гусеничные рычаги, звенья манипулятора и модуль телекамер.

Возможности среды Unity позволяют реализовать импорт трёхмерных моделей в расширении FBX. Таким образом, модель робота, созданная при помощи САПР, может быть использована в Unity. При построении модели была использована САПР SolidWorks. Трёхмерная модель мобильного робота «Капитан» представлена на рисунке 7.

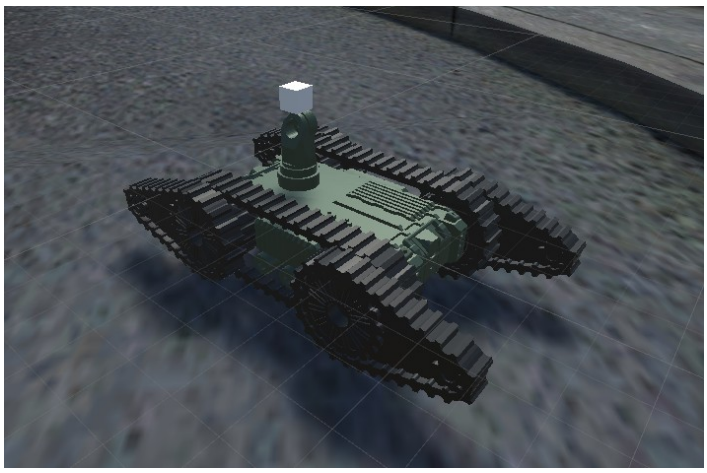

Рисунок 7 – Трёхмерная модель мобильного робота «Капитан»

Взаимодействие составных частей трёхмерной модели в Unity реализовано с помощью двух компонентов: Fixed Joint и Hinge Joint. Компонент Fixed Joint жестко фиксирует объекты друг относительно друга, полностью блокируя как поступательное, так и вращательное движение. При помощи него соединён корпус робота с гусеницами и первое звено манипулятора с модулем телекамер. Hinge Joint реализует вращательную кинематическую пару, допуская только вращательное движение вокруг оси, заданной в локальной системе координат. Таким образом реализовано соединение гусениц робота с гусеничными рычагами, а также вращательные кинематические пары манипулятора.

Габаритные размеры мобильного робота «Капитан» следующие:

- длина: 620 мм,
- ширина: 465 мм,
- высота: 215 мм.

Они соблюдены при создании модели робота в SolidWorks. При импорте модели в Unity линейные размеры не изменяются, однако миллиметры заменяются на единицы измерения Unity. Рекомендовано соответствие одной единицы измерения Unity одному метру, поэтому модель в среде Unity промасштабирована, её линейные размеры уменьшены в тысячу раз.

Перемещение модели робота в виртуальном пространстве реализовано кинематически, без использования сил. Оператор управляет роботом посредством клавиш W, S, A, D, которые соответствуют движению вперёд, назад и повороту против часовой стрелки и по часовой стрелке вокруг вертикальной оси. Алгоритм реализован посредством скрипта на языке C# в среде Unity в файле KLDNcontroller.cs. Программный код файла приведён в приложении 1.

Вращательное движение звеньев манипулятора и гусеничных рычагов модели робота реализовано при помощи возможностей компонента Hinge Joint, который обладает такими элементами как пружина (spring) и демпфер (damper). Пружина стремится достичь желаемого угла (spring.targetPosition), а демпфер ограничивает угловую скорость. Нулевым углом считается угол между звеньями в момент начала симуляции. Алгоритм реализован посредством скрипта на языке C# в среде Unity в файле KLDNcontroller.cs. Программный код файла приведён в приложении А. Перед началом работы элемент spring каждого используемого звена необходимо инициализировать. Это делает функция SpringOn.

Оператор управляет манипулятором позвенно, при помощи клавиш «1» - «8» увеличивая и уменьшая углы между звеньями. Также при помощи клавиш «вверх», «вниз», «влево», «вправо» производится управление гусеничными рычагами.

3.2 Создание моделей камер

Система кругового обзора включает четыре сверхширокоугольные видеокамеры, направленные горизонтально вперёд, назад, вправо и влево, что позволяет в дальнейшем преобразовать изображения в панораму, которая и будет использована для отображения окружающей обстановки на ПДУ. Система с несколькими телекамерами выбрана благодаря высокому разрешению выходного изображения, почти неограниченному полю зрения в

вертикальном направлении и практически полному отсутствию слепых зон [4].

Камеры могут иметь различные параметры и различную конфигурацию расположения на корпусе мобильного робота. Пример одного из вариантов конфигурации приведён на рисунке 11. В данном примере камеры располагаются на блоке, размещенном на первом звене манипулятора. Важно отметить, что такое расположение исключает возможность использования манипулятора.

На изображениях, получаемых с двух соседних камер, присутствуют области, содержащие пиксели, соответствующие одним и тем же точкам пространства. Это необходимое условие для корректной автоматической склейки этих изображений в панораму. Модуль камер представлен в виде куба с длиной стороны 50 мм.

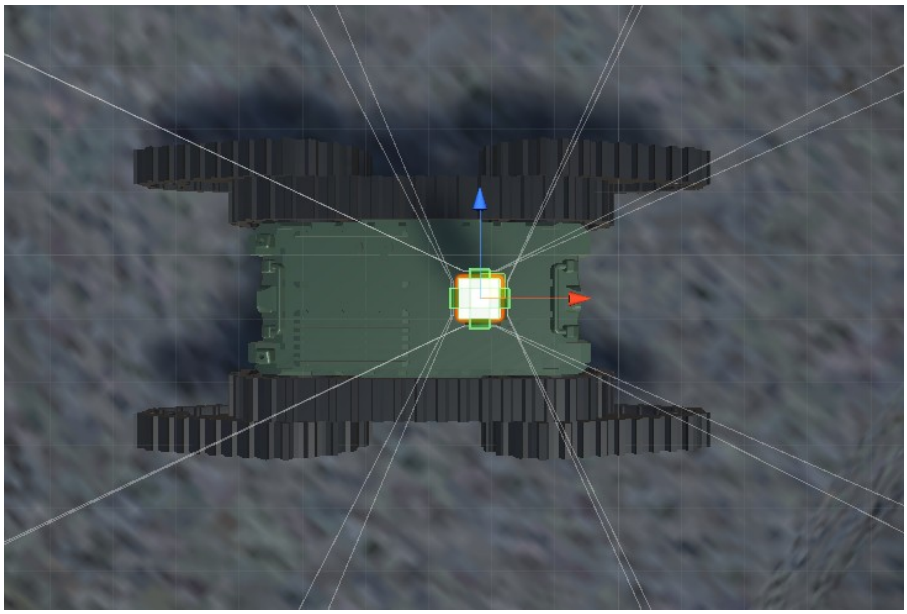

Рисунок 11 – Первая возможная конфигурация расположения камер

Примером другой конфигурации может быть расположение камер не на едином блоке, а непосредственно на корпусе спереди, сзади и по бокам. Расстояние между камерами будет равно соответствующим габаритным размерам робота. Такое решение обычно применяется для автомобилей [5]. Этот вариант расположения камер приведён на рисунке 12.

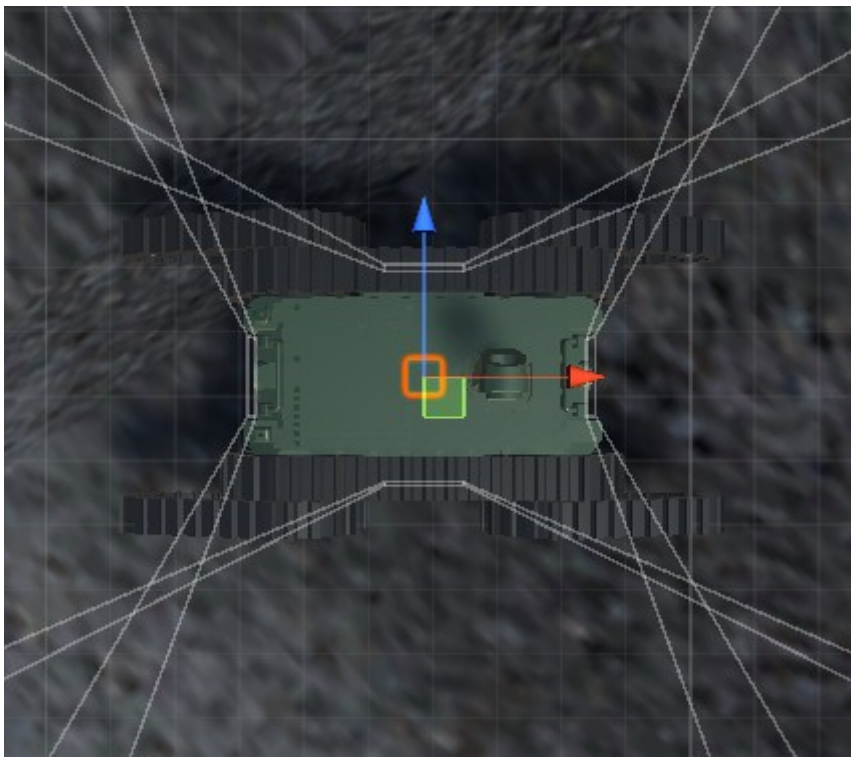

Рисунок 12 – Вторая возможная конфигурация расположения камер

Среда Unity позволяет создавать виртуальные камеры и задавать ряд параметров. При этом виртуальная камера является идеальной, у неё полностью отсутствует дисторсия даже при широком угловом поле зрения. Окно настройки камеры в среде Unity представлено на рисунке 13.

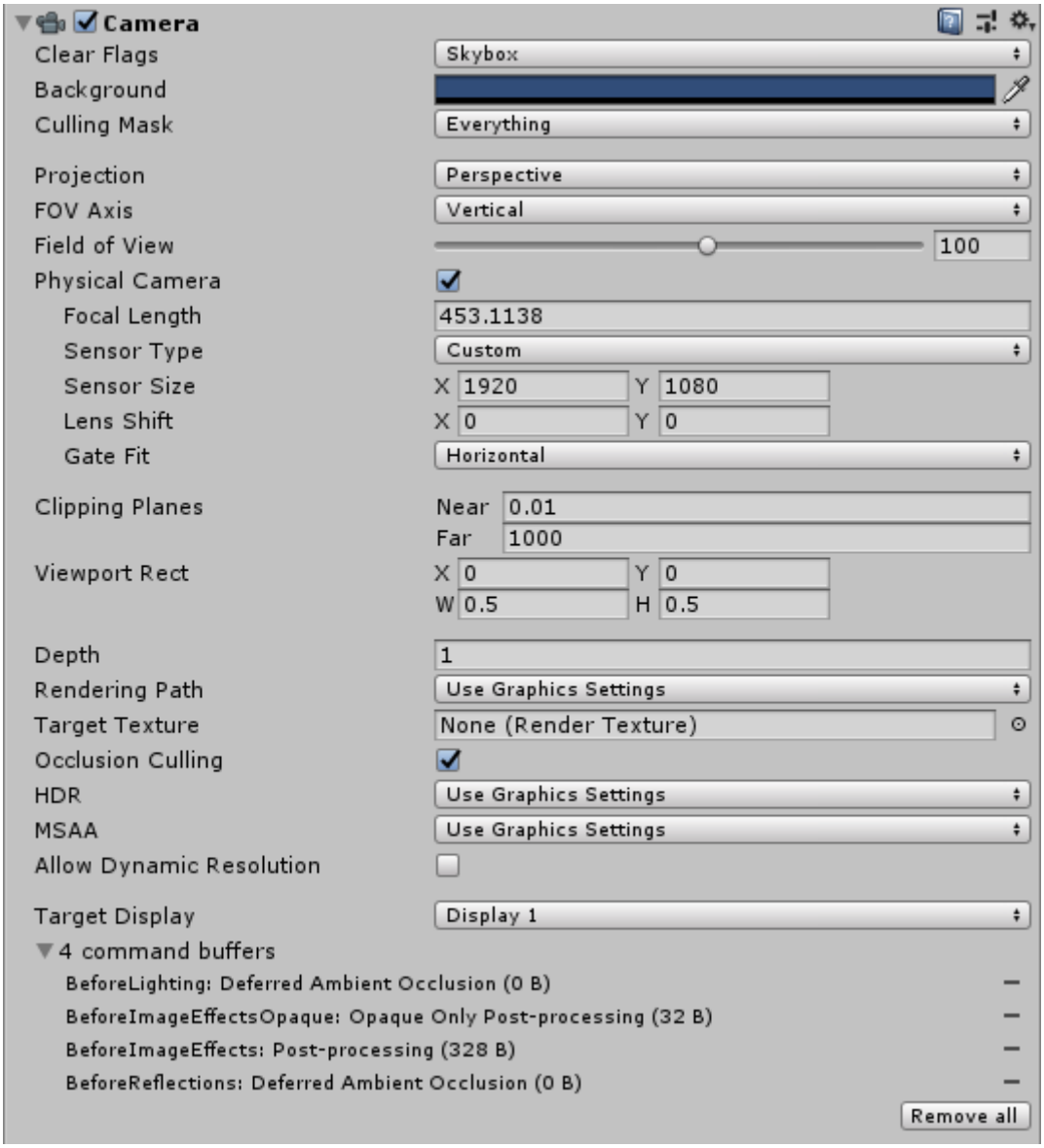

Рисунок 13 – Настройки виртуальной камеры в среде Unity

Группа параметров Transform отвечает за расположение камеры на сцене. К наиболее важным параметрам самой камеры относятся:

- Projection – переключение между перспективным и ортогональным проецированием.

- Field of view – угол обзора камеры в градусах.

- Physical camera – настройка, позволяющая задавать физические параметры камеры, приведённые ниже.

- Focal length – фокусное расстояние.

- Sensor type – формат камеры для симуляции (8 мм, 16 мм и т. п.).

- Sensor size – разрешение матрицы камеры по горизонтальной и вертикальной оси.

- Lens shift – смещение линзы относительно матрицы камеры.

- Clipping planes – минимальное и максимальное расстояние до объектов, попадающий в кадр.

3.3 Создание виртуальной сцены

Основное содержание сцены: плоскость, представляющая собой землю и различные объекты, представляющие опасность для движения робота или служащие ориентирами (автомобили, здания, столбы, дорожные блоки и т.д.). Требованиями к окружающим объектам является наличие характерных точек, которые могут быть использованы при автоматическом объединений изображений с камер в панорамное.

Модели окружающих объектов, как и модель мобильного робота, могут быть созданы с помощью САПР или программ трёхмерного моделирования. Кроме того, Unity позволяет создавать несложные трёхмерные объекты непосредственно в ней. Также данная среда предоставляет множество бесплатных библиотек с трёхмерными моделями, в том числе имитирующими здания, автомобили и дорожное покрытие, которые могут быть использованы в качестве окружающих объектов. Одна из таких библиотек использована при создании объектов обстановки. Виртуальная сцена представлена на рисунке 14.

Рисунок 14 – Виртуальная сцена, составленная из трёхмерных моделей

3.4 Алгоритм передачи изображений для обработки методами технического зрения

В самой среде Unity нет инструментов, позволяющих производить обработку изображений, однако она позволяет использовать плагины, в том числе динамически подключаемые библиотеки (DLL). Обработка изображений с камер производится при помощи DLL, написанной с использованием библиотеки компьютерного зрения OpenCV. В среде Unity на каждом кадре вызывается функция, передающая изображения для обработки функциям OpenCV. После этого можно приступать к дальнейшей обработке.

Связь среды Unity с методами обработки изображений библиотеки OpenCV на данном этапе реализована посредством трёх функций:

- initialize – вызывается в начале симуляции. Принимает на вход размеры изображения в пикселях и вызывает конструкторы массивов cv::Mat, соответствующих входным изображениям.
- getImages – вызывается при обработке каждого кадра. Принимает на вход указатели на массивы, содержащие значения пикселей изображений, а также размеры изображений в пикселях. Заполняет массивы cv::Mat значениями пикселей изображений с виртуальных камер, а также конвертирует цветовую

кодировку и зеркально отражает их. Это связано с тем, что в Unity началом координат изображения является левый нижний пиксель, вертикальная ось направлена вверх и используется порядок цветов RGBA, а в OpenCV началом координат является левый верхний пиксель, вертикальная ось направлена вниз и используется порядок цветов BGR.

- terminate – вызывается при завершении работы симуляции. Высвобождает память и закрывает все окна, открытые при помощи функций OpenCV.

Вышеописанные функции реализованы в пользовательской DLL surroundView, программный код файлов surroundView.h и surroundView.cpp приведён в приложениях Б и В соответственно. На рисунке 15 приведена блок-схема алгоритма функции initialize, где n – число изображений.

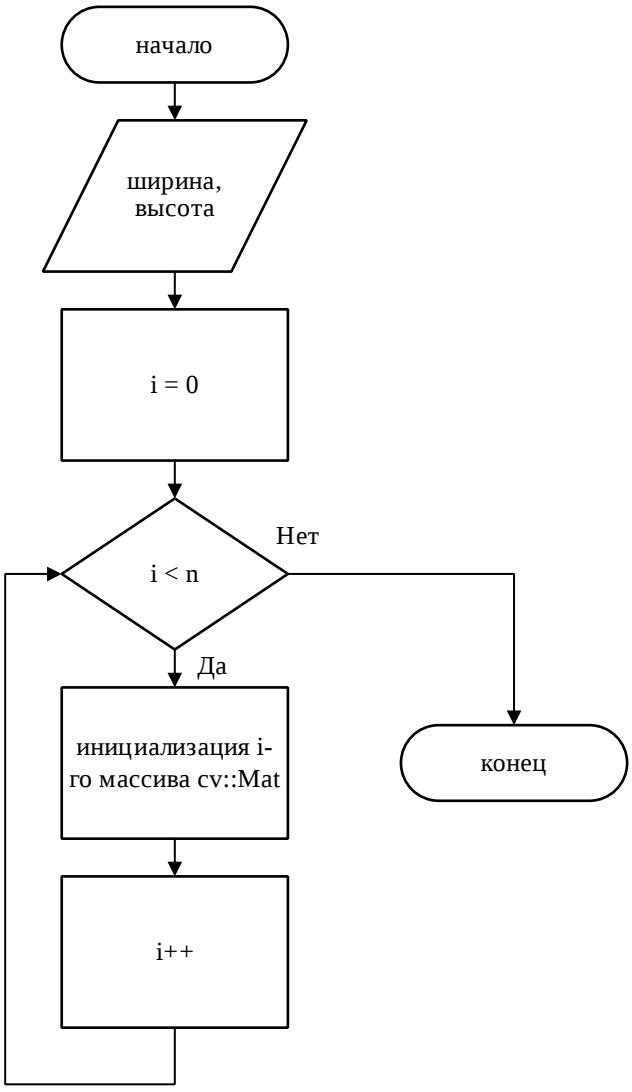

Рисунок 15 – Блок-схема алгоритма функции initialize

На рисунке 16 приведена блок-схема алгоритма функции getImages, где n – число изображений.

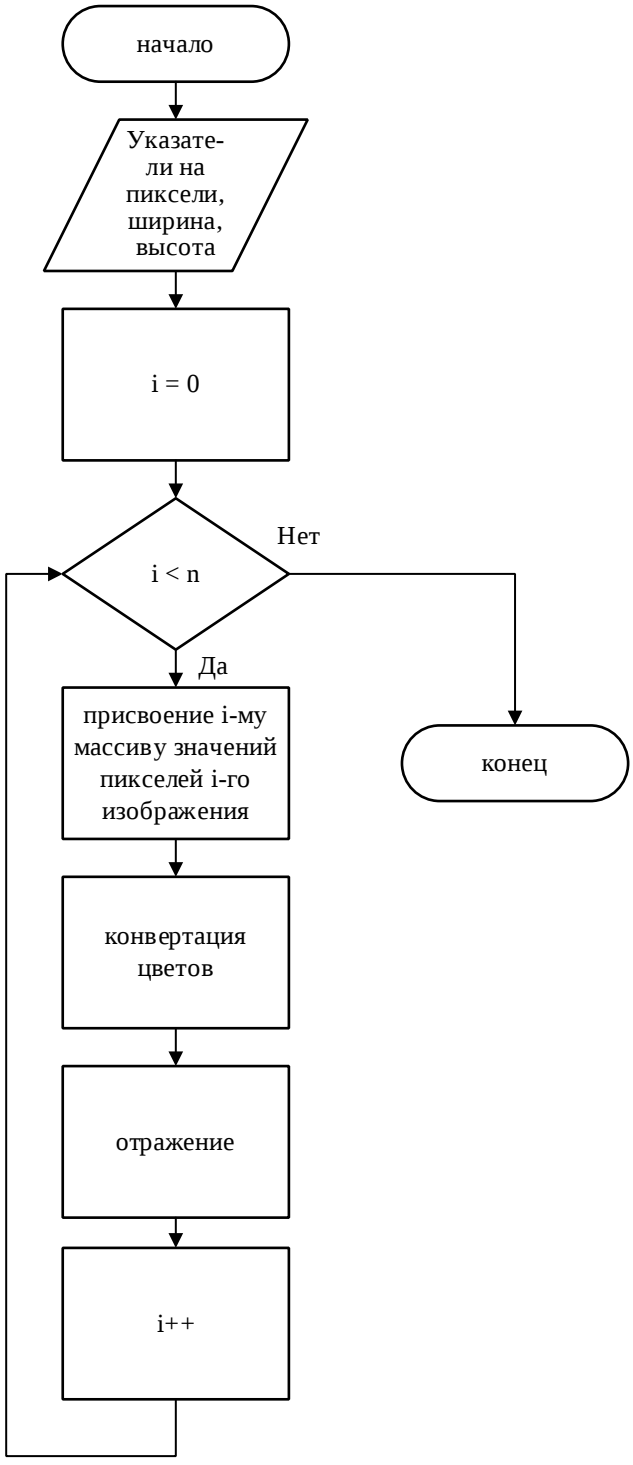

Рисунок 16 – Блок-схема алгоритма функции getImages

На рисунке 17 приведена блок-схема алгоритма функции terminate, где n – число изображений.

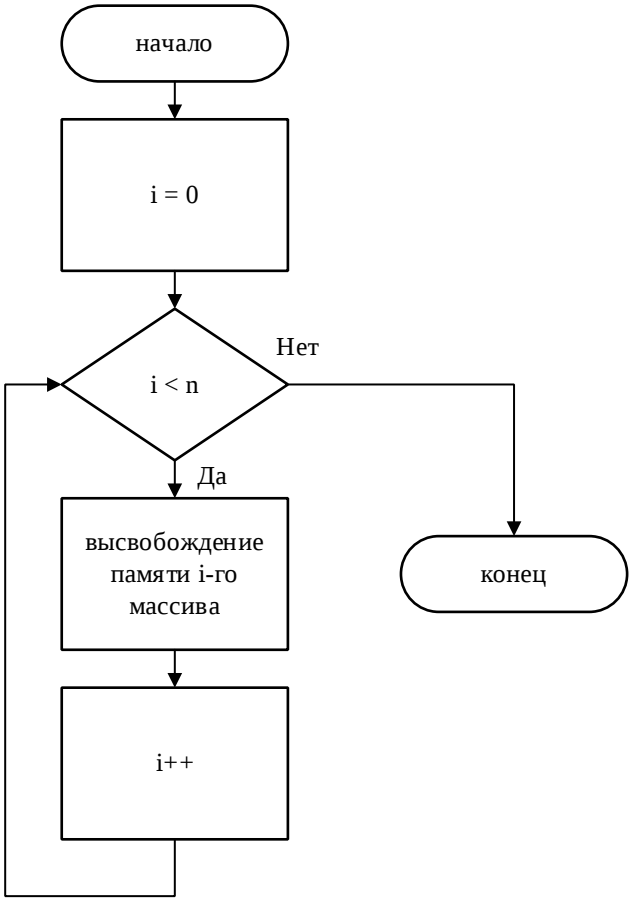

Рисунок 17 – Блок-схема алгоритма функции terminate

В среде Unity изображения преобразовываются при помощи скрипта CamerasManager.cs. Его код представлен в приложении Г. При запуске симуляции вызывается функция initialize, рассмотренная выше. На каждом кадре вызывается функция CameraToTexture2D для каждой камеры. Она принимает на вход виртуальную камеру, преобразует изображение с неё в двумерную текстуру, которую возвращает. Блок-схема алгоритма работы функции CameraToTexture2D представлена на рисунке 18.

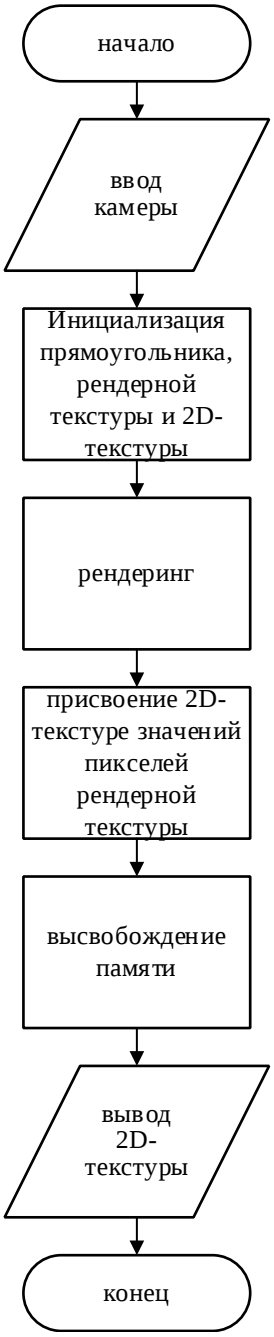

Рисунок 18 – Блок-схема алгоритма функции CameraToTexture2D

После того, как получены двумерные текстуры для всех четырёх камер, вызывается функция TextureToCVMat, которая принимает на вход двумерные текстуры и передаёт их для обработки методами технического зрения. Блок-схема алгоритма работы функции TextureToCVMat представлена на рисунке 19.

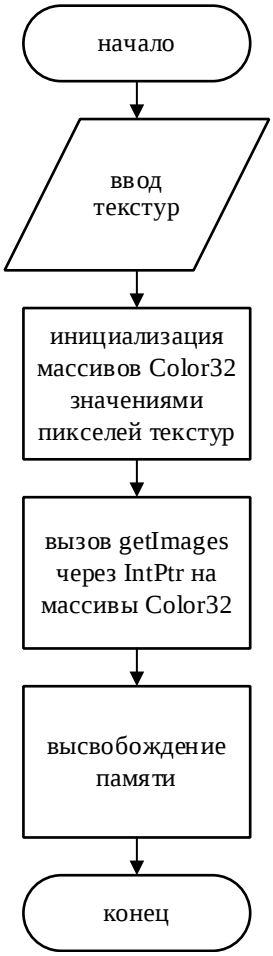

Рисунок 19 – Блок-схема алгоритма функции CameraToTexture2D

При окончании симуляции вызывается функция terminate, рассмотренная выше.

3.5 Выводы по главе

В среде Unity реализована трёхмерная модель мобильного робота «Капитан», модели камер размещены на его корпусе. Реализована виртуальная сцена, включающая в себя трёхмерные модели окружающей обстановки. При помощи скриптов Unity на языке C# реализовано кинематическое перемещение робота по координатам, вращательное движение гусеничных рычагов и звеньев манипулятора.

Реализована передача изображений с виртуальных камер для обработки алгоритмами технического зрения при помощи функций библиотеки OpenCV.

4 Исследование влияния размеров входных изображений на частоту кадров результирующего изображения

4.1 Алгоритм измерения времени работы функций обработки изображений

Важным критерием для оценки быстродействия алгоритмов обработки является время работы функций, связывающих Unity и OpenCV, описанных в предыдущей главе. От времени работы функций напрямую зависит частота кадров выходного изображения. Частота кадров является важным фактором для корректного восприятия оператором окружающей обстановки.

Очевидно, что время обработки напрямую зависит от используемого аппаратного обеспечения. Обработка изображений производится на персональном компьютере с четырёхядерным процессором Intel Core i7-4790K 4001 МГц с графическим процессором NVIDIA GeForce GTX 750. Программное обеспечение реализовано в среде разработки Microsoft Visual Studio 2017 совместно с открытым программным обеспечением в виде библиотеки OpenCV 4.1.1. Используется версия Unity 2019.2.9.

Для вычисления времени работы функций initialize, getImages и terminate можно их модифицировать так, как представлено на блок-схеме на рисунке 20.

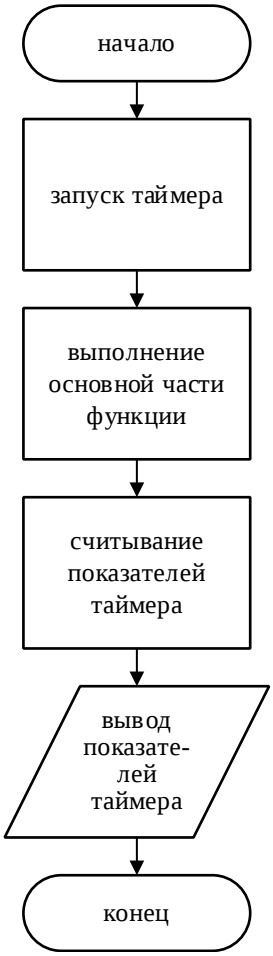

Рисунок 20 – Модифицированный алгоритм функций OpenCV для определения времени их работы

В качестве таймера можно использовать класс OpenCV TickMeter. Он располагает следующими методами для управления отсчётом:

- reset – очищает внутренние значения экземпляра класса,
- start – начинает отсчёт,
- stop – прекращает отсчёт.

Для возвращения значения таймера используются следующие методы:

- getCounter – возвращает внутреннее значение таймера,
- getTimeMicro – возвращает прошедшее время в микросекундах,
- getTimeMilli – возвращает прошедшее время в миллисекундах,
- getTimeSec – возвращает прошедшее время в секундах,
- getTimeTicks – возвращает прошедшее время в тактах таймера.

Значение времени возвращается при помощи типа с плавающей точкой double, однако может впоследствии быть преобразовано в целочисленный тип int. Таким образом, тип возвращаемого значения функций, использующих таймер следует изменить с void на int. Это значение будет возвращено в скрипт Unity, где может быть выведено на дисплей оператора.

Аналогичным образом может быть реализовано измерение времени работы любых функций обработки изображений, которые будут разработаны в дальнейшем. В случае правильной работы функции она возвращает положительное число, равное времени работы, в случае неполадки – отрицательное число, соответствующее ошибке.

Помимо времени, затраченного на обработку изображений функциями OpenCV, в Unity можно определить текущую частоту кадров через время, затрачиваемое на обработку одного кадра. Частота кадров отображается в левом верхнем углу конечного изображения. Вывод на дисплей реализован с помощью функции OnGUI в скрипте KLDNController.cs. Блок-схема алгоритма её работы представлена на рисунке 21.

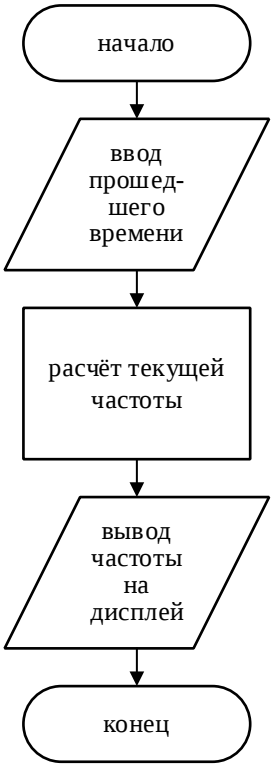

Рисунок 21 – Алгоритм функции вывода текущей частоты на дисплей

Аналогичным способом может быть выведено на дисплей время работы каждой функции, связывающей Unity с библиотекой OpenCV, возвращенное этой функцией. Также существует возможность просмотреть возвращенное значение путём создания публичной глобальной переменной в Unity, которая будет отображать текущее время работы функций.

4.2 Измерение времени работы функций обработки изображений

С помощью описанных выше алгоритмов измерим среднее время работы функции initialize при различных разрешениях входного изображения. Результаты измерений представлены в таблице 2.

Таблица 2 – Результаты измерения времени работы функции initialize при различных разрешениях входного изображения

Разрешение, пиксели	Среднее время работы, мкс
480 x 270	27
960 x 540	17
1920 x 1080	17

Видно, что корреляция между размером входных изображений и временем инициализации массивов практически отсутствует. Это связано с тем, что операция инициализации только выделяет память под массивы, но не заполняет их.

С помощью описанных выше алгоритмов измерим среднее время работы функции getImages при различных разрешениях входного изображения. Результаты измерений представлены в таблице 3 и на рисунке 22.

Таблица 3 – Результаты измерения времени работы функции getImages при различных разрешениях входного изображения

Разрешение, пиксели	Среднее время работы, мкс
480 x 270	512
960 x 540	5650

1920 x 1080	26352

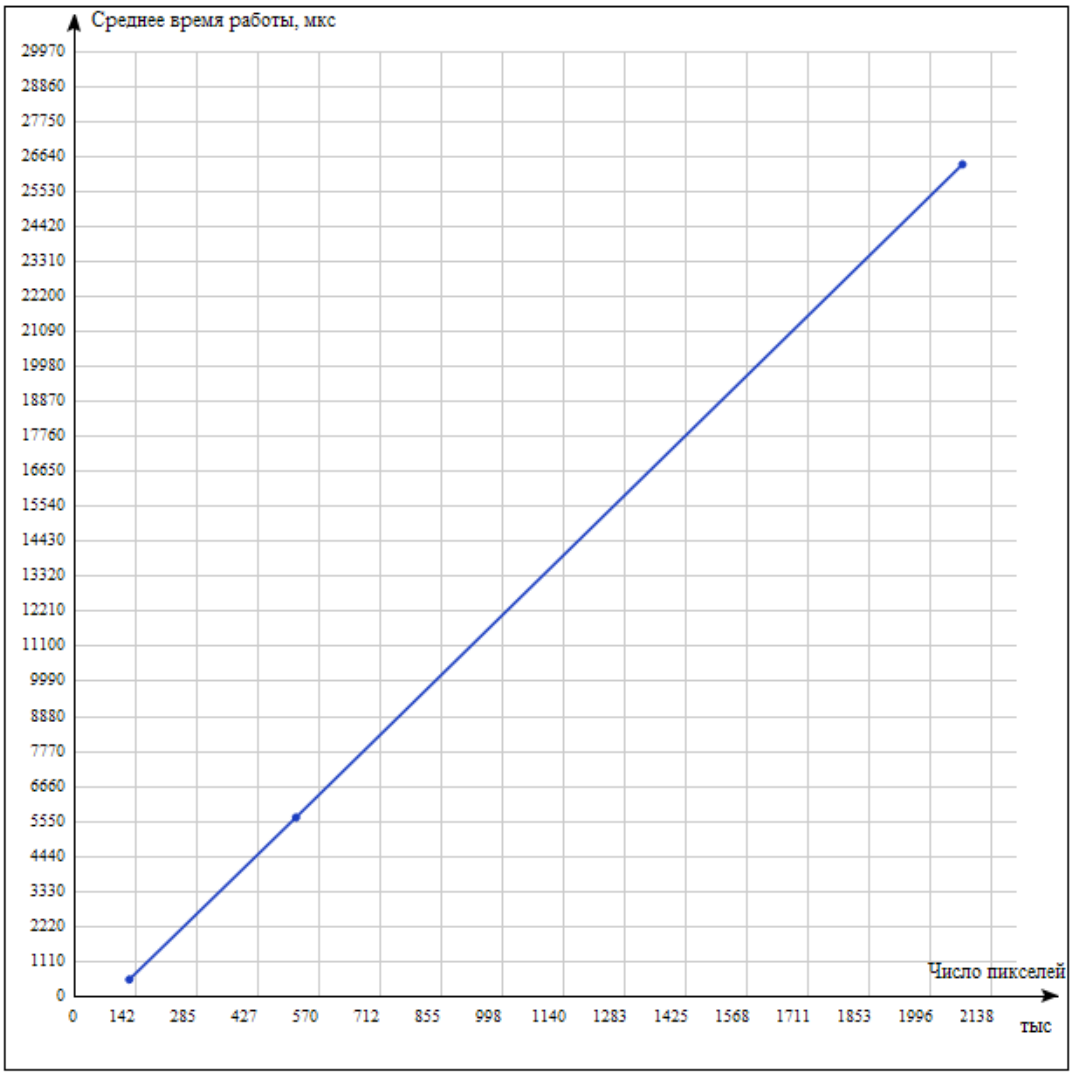

Рисунок 22 – Результаты измерения времени работы функции getImages при различных разрешениях входного изображения

Очевидно, что время работы функции прямо пропорционально числу пикселей во входных изображениях, поскольку в ходе её работы все они заполняются новыми значениями.

С помощью описанных выше алгоритмов измерим время работы функции terminate при различных разрешениях входного изображения. Результаты измерений представлены в таблице 4.

Таблица 4 – Результаты измерения времени работы функции terminate при различных разрешениях входного изображения

Разрешение, пиксели	Среднее время работы, мкс

480 x 270	4
960 x 540	536
1920 x 1080	2027

Из результатов измерения видно, что в отличие от функции инициализации, функция очистки памяти зависит от размеров изображений, поскольку необходимо очистить значение, соответствующее каждому пикселю изображения.

Также измерим частоту кадров выходного изображения в Unity способом, описанным выше. Результаты измерений представлены в таблице 5.

Таблица 5 – Результаты измерения частоты кадров выходного изображения при различных разрешениях входного изображения

Разрешение, пиксели	Частота, Гц
480 x 270	58
960 x 540	36
1920 x 1080	6

Важно заметить, что частота выходного изображения определяется в данном случае в том числе работой функций Unity.

4.3 Выводы по главе

Разработан алгоритм измерения времени работы функций обработки изображений при помощи библиотеки OpenCV с дальнейшей передачей данных о времени работы в Unity и выводом на дисплей оператора.

Проведены измерения времени работы функций initialize, getImages и terminate с целью выявления зависимости времени работы от разрешения передаваемых изображений. Выяснено, что время инициализации массивов, соответствующих изображениям, не зависит от их разрешения. Время передачи изображений и время очистки памяти прямо пропорционально числу пикселей входного изображения.

ЗАКЛЮЧЕНИЕ

В ходе работы рассмотрен мировой опыт использования телевизионных систем мобильных роботов, в частности, систем кругового

обзора, а также опыт их виртуального моделирования. Выбрано программное обеспечение для разработки: среда разработки Unity, библиотека OpenCV и интегрированная среда разработки Visual Studio. Рассмотрены математические модели используемых телекамер и виды преобразований изображений.

Виртуальная модель системы кругового обзора создана с использованием среды разработки Unity. Рассмотрены возможные варианты расположения телекамер системы кругового обзора. Создана соответствующая требованиям модель мобильного робота. Установлены и настроены виртуальные телекамеры. Реализована передача изображений с телекамер для обработки алгоритмами компьютерного зрения. Исследована зависимость между разрешением входных изображений и частотой кадров выходного изображения.

В дальнейшем предполагается продолжить отработку алгоритмов с использованием созданной модели. Ниже приведены примеры алгоритмов, которые могут быть отработаны:

- Различные варианты объединения изображений с камер в панорамное (цилиндрическое или сферическое, с ручной настройкой параметров или с их автоматическим расчётом);
- Гомографическое преобразование изображений с их объединением в вид сверху;
- Проецирование изображения на разные виды поверхностей (параболоидную, сферическую и другие) для получения вида «от третьего лица»;
- Реализация возможности изменения угла обзора оператора на выходном изображении.

Кроме того, имеет смысл реализовать возможность изменения угла наклона камер относительно горизонтальной оси, позволяя оператору изменять число пикселей, соответствующих ближним или дальним точкам

окружающего пространства, таким образом получая более детальное представление о наиболее интересующей его в данный момент области.

Сперва стоит реализовать эту возможность программно на виртуальной модели, а в дальнейшем разработать универсальный модуль, позволяющий регулировать наклон камер.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Суворов К. А. Системы виртуальной реальности и их применение // T-Comm. 2013. №9.

2 Cooper J., Vik J.O., Waltemath D. A call for virtual experiments: accelerating the scientific process // PeerJ PrePrints 2:e273v1. 2014.

3 В.В. Варлашин, Алгоритмическое обеспечение системы кругового обзора для мобильных робототехнических комплексов. 2018.

4 S. Schulter, P. Vernaza, M. Chandraker, M. Zhai, Generating occlusion-aware bird eye view representations of complex road scenes // U.S. Patent US20190096125A1. 2019.

5 V. Gojak, J. Janjatovic, N. Vukota, M. Milosevic, M. Z. Bjelica, Informational Bird's Eye View System for Parking Assistance // 2017 IEEE 7th International Conference on Consumer Electronics - Berlin (ICCE-Berlin), 2017

6 И.В. Валиев, А.Г. Волобой, Моделирование монитора кругового обзора // Институт прикладной математики им. М.В.Келдыша РАН, Москва, 2010.

7 Y. Song, Q. Shi, Q. Huang and T. Fukuda. Development of an omnidirectional vision system for environment perception // IEEE International Conference on Robotics and Biomimetics (ROBIO 2014), Bali, 2014, pp. 695-700.

8 Qing Shi, Chang Li, Chunbao Wang, Haibo Luo, Qiang Huang, Toshio Fukuda. Design and implementation of an omnidirectional vision system for robot perception // Mechatronics, Volume 41, 2017, pp 58-66.

9 D.S. Popov, O.A. Shmakov, Requirements for remote control systems for ground-based mobile robots // 30th International Scientific and Technological Conference «Extreme Robotics-2019», 2019

10 Iris. [Электронный ресурс] URL: http://www.robo-team.com/products/iris/ (дата обращения: 27.11.2019)

11 Scarabaeus. [Электронный ресурс] URL: https://www.set-1.ru/products/robototekhnicheskie-kompleksy-/skarabey/ (дата обращения: 27.11.2019)

12 Nerva. [Электронный ресурс] URL: https://www.unhitec-robotics.com/blank. (дата обращения: 28.11.2019)

13 МРК «Капитан». [Электронный ресурс] URL: https://rtc.ru/media/documents/КАПИТАН_брошюра_ 10.18.pdf. (дата обращения: 28.11.2019)

14 Bradski, G. and Kaehler, A., Learning Open CV: Computer Vision with the OpenCV Library // O'Reilly, 2008.

15 A. Jutamanee & H. Xu, Fish-eye image of hemi-cylinder unwrapping plane based on a flexible technique // 2015 International Conference on Fluid Power and Mechatronics, 2015

16 T. Sato, A. Moro, A. Sugahara, T. Tasaki, A. Yamashita, H. Asama, Spatio-Temporal Bird's-eye View Images Using Multiple Fish-eye Cameras // 2013 IEEE/SICE International Symposium on System Integration, 2013

17 OpenCV: Fisheye camera model. [Электронный ресурс] URL: https://docs.opencv.org/3.4/db/d58/group__calib3d__fisheye.html. (дата обращения: 28.11.2019)

18 R. Szeliski, Computer Vision: Algorithms and Applications // Springer, 2011

19 Прудников Н.В., Шлишевский В.Б. Панорамные оптико-электронные устройства кругового и секторного обзора // Вестник СГУГиТ, вып. 1 (33). 2016. С. 148-161.

20 A. Jutamanee, H. Xu, W. Zhao, Development of Hemi-Cylinder Plane for Panorama View in Around View Monitor Applications // 2016 International Conference on Computational Intelligence and Applications, 2016

21 Лазаренко В.П., Джамийков Т.С., Коротаев В.В., Ярышев С.Н. Метод создания сферических панорам из изображений, полученных всенаправленными оптико-электронными системами // Научно-технический вестник информационных технологий, механики и оптики, Т. 16. № 1. СПБ: ИТМО, 2016. С. 46–53.

Приложение А. Программный код файла KLDNcontroller.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class KLDNcontroller : MonoBehaviour
{
    [Header("Levers")]
    public HingeJoint LeverLF;
    public HingeJoint LeverRF;
    public HingeJoint LeverLB;
    public HingeJoint LeverRB;

    public float Velocity = 20f;
    public float moveSpeed = 10f;
    public float turnSpeed = 50f;
    private Vector3 currentRotation;
    private Vector3 direction;

    [Header("Links")]
    public HingeJoint Link1;
    public HingeJoint Link2;
    public HingeJoint Link3;
    public HingeJoint Link4;

    [Header("Wheels")]

    public WheelCollider WheelLF;
    public WheelCollider WheelRF;
    public WheelCollider WheelLB;
    public WheelCollider WheelRB;

    // Start is called before the first frame update
    void Start()
    {
```

```
        SpringOn();
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        #region levers
        if (Input.GetKey("up"))
        {
            JointSpring SpringLF = LeverLF.spring;
            SpringLF.targetPosition = SpringLF.targetPosition +
Time.fixedDeltaTime * Velocity;
            LeverLF.spring = SpringLF;

            JointSpring SpringRF = LeverRF.spring;
            SpringRF.targetPosition = SpringRF.targetPosition +
Time.fixedDeltaTime * Velocity;
            LeverRF.spring = SpringRF;
        }

        if (Input.GetKey("down"))
        {
            JointSpring SpringLF = LeverLF.spring;
            SpringLF.targetPosition = SpringLF.targetPosition -
Time.fixedDeltaTime * Velocity;
            LeverLF.spring = SpringLF;

            JointSpring SpringRF = LeverRF.spring;
            SpringRF.targetPosition = SpringRF.targetPosition -
Time.fixedDeltaTime * Velocity;
            LeverRF.spring = SpringRF;
        }

        if (Input.GetKey("left"))
        {
            JointSpring SpringLB = LeverLB.spring;
            SpringLB.targetPosition = SpringLB.targetPosition -
Time.fixedDeltaTime * Velocity;
            LeverLB.spring = SpringLB;

            JointSpring SpringRB = LeverRB.spring;
            SpringRB.targetPosition = SpringRB.targetPosition -
Time.fixedDeltaTime * Velocity;
            LeverRB.spring = SpringRB;
        }

        if (Input.GetKey("right"))
        {
            JointSpring SpringLB = LeverLB.spring;
            SpringLB.targetPosition = SpringLB.targetPosition +
Time.fixedDeltaTime * Velocity;
            LeverLB.spring = SpringLB;
```

```
            JointSpring SpringRB = LeverRB.spring;
            SpringRB.targetPosition = SpringRB.targetPosition +
Time.fixedDeltaTime * Velocity;
            LeverRB.spring = SpringRB;
        }
        #endregion levers

        #region links
        if (Input.GetKey("1"))
        {
            JointSpring Spring1 = Link1.spring;
            Spring1.targetPosition = Spring1.targetPosition +
Time.fixedDeltaTime * Velocity;
            Link1.spring = Spring1;
        }

        if (Input.GetKey("2"))
        {
            JointSpring Spring1 = Link1.spring;
            Spring1.targetPosition = Spring1.targetPosition -
Time.fixedDeltaTime * Velocity;
            Link1.spring = Spring1;
        }

        if (Input.GetKey("3"))
        {
            JointSpring Spring2 = Link2.spring;
            Spring2.targetPosition = Spring2.targetPosition +
Time.fixedDeltaTime * Velocity;
            Link2.spring = Spring2;
        }

        if (Input.GetKey("4"))
        {
            JointSpring Spring2 = Link2.spring;
            Spring2.targetPosition = Spring2.targetPosition -
Time.fixedDeltaTime * Velocity;
            Link2.spring = Spring2;
        }

        if (Input.GetKey("5"))
        {
            JointSpring Spring3 = Link3.spring;
            Spring3.targetPosition = Spring3.targetPosition +
Time.fixedDeltaTime * Velocity;
            Link3.spring = Spring3;
        }

        if (Input.GetKey("6"))
        {
            JointSpring Spring3 = Link3.spring;
```

```
            Spring3.targetPosition = Spring3.targetPosition -
Time.fixedDeltaTime * Velocity;
            Link3.spring = Spring3;
        }

        if (Input.GetKey("7"))
        {
            JointSpring Spring4 = Link4.spring;
            Spring4.targetPosition = Spring4.targetPosition +
Time.fixedDeltaTime * Velocity;
            Link4.spring = Spring4;
        }

        if (Input.GetKey("8"))
        {
            JointSpring Spring4 = Link4.spring;
            Spring4.targetPosition = Spring4.targetPosition -
Time.fixedDeltaTime * Velocity;
            Link4.spring = Spring4;
        }
        #endregion links

        #region motors
        int w = Input.GetKey("w") ? 1 : 0;
        int a = Input.GetKey("a") ? 1 : 0;
        int s = Input.GetKey("s") ? -1 : 0;
        int d = Input.GetKey("d") ? -1 : 0;

        float move = (w + s);
        float turn = (a + d);

        transform.Translate(move * direction * moveSpeed *
Time.deltaTime, Space.World);

        transform.Rotate(turn * Vector3.up, -turnSpeed *
Time.deltaTime);

        currentRotation = transform.rotation.eulerAngles;
        direction = new Vector3(Mathf.Sin(Mathf.Deg2Rad *
currentRotation.y), 0, Mathf.Cos(Mathf.Deg2Rad *
currentRotation.y));
        #endregion motors
    }

    public void SpringOn()
    {
        JointSpring jointSpring = new JointSpring
        {
            damper = 10000,
            spring = 1000000,
            targetPosition = 0
        };
```

```
        LeverLF.spring = jointSpring;

        jointSpring.targetPosition = LeverRF.angle;
        LeverRF.spring = jointSpring;

        jointSpring.targetPosition = LeverLB.angle;
        LeverLB.spring = jointSpring;

        jointSpring.targetPosition = LeverRB.angle;
        LeverRB.spring = jointSpring;

        jointSpring.targetPosition = Link1.angle;
        Link1.spring = jointSpring;

        jointSpring.targetPosition = Link2.angle;
        Link2.spring = jointSpring;

        jointSpring.targetPosition = Link3.angle;
        Link3.spring = jointSpring;

        jointSpring.targetPosition = Link4.angle;
        Link4.spring = jointSpring;

        LeverLF.useSpring = true;
        LeverRF.useSpring = true;
        LeverLB.useSpring = true;
        LeverRB.useSpring = true;
        Link1.useSpring = true;
        Link2.useSpring = true;
        Link3.useSpring = true;
        Link4.useSpring = true;
    }

    void OnGUI()
    {
        GUI.Label(new Rect(0, 0, 100, 100), (1.0f /
(Time.smoothDeltaTime)).ToString());
    }

}
```

Приложение Б. Программный код файла surroundView.h

```
#include "stdafx.h"
#include <opencv2/opencv.hpp>

struct Color32
{
     uchar r;
     uchar g;
```

```
	uchar b;
	uchar a;
};

extern "C"
{
	__declspec(dllexport) void initialize(int width, int
height);
	__declspec(dllexport) void getImages(Color32* raw1,
Color32* raw2, Color32* raw3, Color32* raw4, int width, int
height);
	__declspec(dllexport) void terminate();
}
```

Приложение В. Программный код файла surroundView.cpp

```
#include "stdafx.h"
#include <opencv2/opencv.hpp>
#include "surroundView.h"

using namespace cv;
using namespace std;

extern "C"
{
	Mat frame[4];


	void initialize(int width, int height)
	{
		for (uint8_t i = 0; i < 4; i++)
			frame[i] = Mat(height, width, CV_8UC4);
	}

	void getImages(Color32* raw1, Color32* raw2, Color32* raw3,
Color32* raw4, int width, int height)
	{
		frame[0] = Mat(height, width, CV_8UC4, raw1);
		frame[1] = Mat(height, width, CV_8UC4, raw2);
		frame[2] = Mat(height, width, CV_8UC4, raw3);
		frame[3] = Mat(height, width, CV_8UC4, raw4);
		for (uint8_t i = 0; i < 4; i++)
		{
			cvtColor(frame[i], frame[i], COLOR_BGRA2RGB);
			flip(frame[i], frame[i], 0);
		}
	}

	void terminate()
	{
		for (uint8_t i = 0; i < 4; i++)
```

```
            {
                frame[i].release();
            }
            destroyAllWindows();
        }
}
```

Приложение Г. Программный код файла CamerasManager.cs

```
using UnityEngine;
using UnityEngine.UI;
using System;
using System.Collections;
using System.Runtime.InteropServices;

public class CamerasManager : MonoBehaviour
{
    [DllImport("surroundView", EntryPoint = "initialize")]
    unsafe private static extern void initialize(int width, int
height);

    [DllImport("surroundView", EntryPoint = "getImages")]
    unsafe private static extern void getImages(IntPtr raw1,
IntPtr raw2, IntPtr raw3, IntPtr raw4, int width, int height);

    [DllImport("surroundView", EntryPoint = "terminate")]
    unsafe private static extern void terminate();

    public int width;
    public int height;

    private Texture2D CameraToTexture2D(Camera camera)
    {
        Rect rect = new Rect(0, 0, width, height);
        RenderTexture renderTexture = new RenderTexture(width,
height, 24);
        Texture2D screenShot = new Texture2D(width, height,
TextureFormat.ARGB32, false);

        camera.targetTexture = renderTexture;
        camera.Render();

        RenderTexture.active = renderTexture;
        screenShot.ReadPixels(rect, 0, 0);

        camera.targetTexture = null;
        RenderTexture.active = null;

        Destroy(renderTexture);
        renderTexture = null;
        return screenShot;
```

```
    }

    unsafe void TextureToCVMat(Texture2D raw1, Texture2D raw2,
Texture2D raw3, Texture2D raw4)
    {
        Color32[] rawColor1 = raw1.GetPixels32();
        Color32[] rawColor2 = raw2.GetPixels32();
        Color32[] rawColor3 = raw3.GetPixels32();
        Color32[] rawColor4 = raw4.GetPixels32();

        fixed (Color32* p1 = rawColor1, p2 = rawColor2, p3 =
rawColor3, p4 = rawColor4)
        {
            getImages((IntPtr)p1, (IntPtr)p2, (IntPtr)p3,
(IntPtr)p4, raw1.width, raw1.height);
        }
        Destroy(raw1);
        Destroy(raw2);
        Destroy(raw3);
        Destroy(raw4);
        raw1 = null;
        raw2 = null;
        raw3 = null;
        raw4 = null;
    }

    public Camera camera1;
    public Camera camera2;
    public Camera camera3;
    public Camera camera4;

    private Texture2D camTex1;
    private Texture2D camTex2;
    private Texture2D camTex3;
    private Texture2D camTex4;

    private Rect readingRect;

    void Start()
    {
        initialize(width, height);
    }

    void Update()
    {
        camTex1 = CameraToTexture2D(camera1);
        camTex2 = CameraToTexture2D(camera2);
        camTex3 = CameraToTexture2D(camera3);
        camTex4 = CameraToTexture2D(camera4);

        TextureToCVMat(camTex1 ,camTex2, camTex3, camTex4);
    }
```

```
    void OnApplicationQuit()
    {
        terminate();
    }
}
```