

Список используемых сокращений, обозначений, терминов

БНТС – беспилотное наземное транспортное средство.

Сегментация – процесс присвоения каждому пикселью на исходном изображении значения того или иного класса, связанного с определённым объектом интереса на нём и имеющего свой цвет, в результате которого исходное изображение превращается в сегментированное – его цветовую маску, которая впоследствии программно обрабатывается другими алгоритмами.

YOLO (You Only Look Once) – название модели свёрточной нейронной сети глубокого обучения для детектирования объектов в режиме реального времени.

ROS (Robot Operating System) – операционная система для роботов. Название основного фреймворка (его второй версии), используемого при разработке решения.

RGBD-изображение – цветное трёхканальное RGB-изображение с дополнительным каналом так называемой глубины ([англ.](#) depth – D), который содержит в себе информацию о расстоянии от видеокамеры до каждого из объектов в поле её видимости на изображении.

Эго-автомобиль – виртуальный беспилотный автомобиль, участвующий в испытаниях.

Углы Эйлера – три угла поворота по трём соответствующим осям координат X, Y и Z, определяющие ориентацию объекта в пространстве путём его последовательного вращения вокруг них.

Кватернион – математический объект, хранящий в себе информацию о вращении и ориентации в трёхмерном пространстве другого объекта вокруг соответствующих осей координат X, Y и Z на заданное значение угла W.

BEV (Bird's-Eye View) – вид с высоты птичьего полёта.

DDS (Data Distribution Service) – протокол обмена сообщениями в режиме реального времени между узлами ROS 2, основанный на модели публикации-подписки.

SLAM (Simultaneous Localization and Mapping) – набор методов для одновременной локализации и построения карты неизвестного окружения.

Детекция – процесс выделения каждого объекта интереса на исходном изображении в так называемый ограничивающий прямоугольник или рамку (англ. *bounding box*) с известным набором координат для его дальнейшей программной обработки.

AMCL (Adaptive Monte Carlo Localization) – метод адаптивной локализации Монте-Карло для определения вероятного месторасположения робота на известной карте.

Дисторсия – искажение (ошибка или погрешность) изображения в оптической системе, вызванное отклонением по тем или иным причинам (например, повреждение линзы объектива) светового луча от того направления, по которому он должен был бы идти в идеальной оптической системе.

DEF-наименование – уникальное имя объекта на сцене мира в робототехническом 3D-симуляторе Webots, используемое для его идентификации и отличия от других таких узлов (нод).

Гомография (гомографическое преобразование) – вид геометрического преобразования на одной плоскости, которое отображает точки одного изображения в точно такие же точки соответствия на втором изображении, похожем на первое.

Фрейм – именованная система координат, описывающая положение и ориентацию того или иного объекта в пространстве. Связь между несколькими фреймами может задаваться как вручную, через так называемые трансформации, с использованием временных меток, так и автоматически, считываясь, например, из файла с описанием всех звеньев робота.

Ковариационная матрица (ковариация) – математический объект, который используется для описания неопределённости тех или иных измерений (например, угловой скорости движения) и описывает степень взаимной зависимости между компонентами их случайных векторов.

Эвристика – оценочная функция, используемая для приближённого решения задач, особенно в ситуациях, когда данных недостаточно. Зачастую основывается на опыте, здравом смысле, интуиции или эмпирических наблюдениях.

NFP (Navigation Function Planner) – алгоритм планирования глобального пути движения NavFn Planner из фреймворка для автономной навигации Nav2.

RPP (Regulated Pure Pursuit) – алгоритм следования локальному пути движения из Nav2.

MPPI (Model Predictive Path Integral) – контроллер следования локальному пути движения оттуда же.

ТЕВ (Timed Elastic Band) – алгоритм планирования локального пути движения, учитывающий кинематические ограничения робота и препятствия вокруг него.

MPC (Model Predictive Control) – метод управления, при котором оптимальное управляющее воздействие рассчитывается с учётом модели системы и предсказания её будущих состояний.

BEV-изображение – полученное вследствие гомографического преобразования изображение с видом на свою же перспективную версию, но с высоты птичьего полёта (вид сверху).

CVAT (Computer Vision Annotation Tool) – инструмент разметки данных в области компьютерного зрения. Частично бесплатное веб-приложение, которое позволяет человеку вручную производить сегментацию и детекцию объектов интереса на входных изображениях из заранее собранного датасета, после чего экспортовать эту разметку в требуемом для обучения модели глубокого обучения формате.

Датасет – набор обучающих, валидационных и тестовых данных, которые предварительно собираются для обучения нейронной сети и могут представлять из себя не только изображения, но также и текст, аудио или вообще всё одновременно.

Оглавление

Введение.....	10
1 Постановка задачи и анализ предметной области.....	12
1.1 Проблема.....	12
1.2 Цели и задачи.....	13
1.3 Обобщённая постановка задачи.....	14
1.4 Расширенная постановка задачи.....	15
1.4.1 Потоки обработки данных.....	15
1.4.2 Функциональные требования.....	17
1.4.3 Нефункциональные требования.....	18
1.5 Анализ предметной области.....	20
1.5.1 Анализ существующих подходов.....	20
1.5.2 Описание используемых технологий и инструментов.....	25
1.5.3 Обоснование выбора технологической платформы.....	28
2 Проектирование и реализация решения.....	30
2.1 Общая архитектура.....	30
2.2 Калибровка виртуальных видеокамер.....	33
2.3 Система двухмерного кругового обзора.....	45
2.4 Сегментированная локальная карта.....	56
2.5 Глобальная карта и планирование маршрута движения беспилотного наземного транспортного средства.....	60
2.5.1 Глобальная карта.....	60
2.5.2 Планирование маршрута движения БНТС.....	75
NavFn Planner.....	76
Regulated Pure Pursuit.....	81
Model Predictive Path Integral.....	83
3 Тестирование решения.....	85
3.1 Калибровка виртуальных видеокамер.....	86
3.1.1 Эксперимент №1.....	86
3.1.2 Эксперимент №2.....	92

3.2 Система двухмерного кругового обзора.....	95
3.2.1 Эксперимент №1	95
3.2.2 Эксперимент №2.....	102
3.3 Сегментированная локальная карта.....	105
3.3.1 Эксперимент №1.....	105
3.4 Глобальная карта и планирование маршрута движения беспилотного наземного транспортного средства.....	112
3.4.1 Глобальная карта.....	112
3.4.1.1 Эксперимент №1.....	112
3.4.2 Планирование маршрута движения БНТС.....	118
3.4.2.1 Эксперимент №1.....	118
3.4.2.2 Эксперимент №2.....	122
3.4.2.3 Эксперимент №3.....	123
Заключение.....	128
Список используемой литературы.....	129
Приложение А. Листинги программного кода.....	135

Введение

В последние годы, технологии автономного управления стремительно развиваются, находя широкое применение в различных отраслях, включая промышленность, логистику и городской транспорт. Беспилотные наземные транспортные средства (далее – *БНТС*) являются довольно перспективным направлением в области робототехники с миллиардными инвестициями не только в Российской Федерации и странах СНГ [1], но и по всему миру [2], привлекая к себе внимание всё большего числа людей, причём не только рядовых студентов и исследователей, но и более опытных разработчиков и инженеров крупных компаний.

Одним из элементов или даже концепцией при разработке *БНТС* является их восприятие окружающего пространства, которое служит основой для безопасного и эффективного передвижения подобного рода транспортных средств как по просторным и безлюдным складским помещениям, так и по тесным городским улицам, переполненным людьми.

Для решения этой задачи используются различные типы сенсоров, включая радары, видеокамеры и лидары, однако традиционные методы картирования зачастую требуют значительных вычислительных ресурсов, а также не всегда обеспечивают достаточный уровень детализации в режиме реального времени [3], что является критической проблемой для любого вида беспилотного транспорта, управляет которым большую часть времени не человек, а компьютер.

Одним из возможных решений данной проблемы является использование системы двухмерного кругового обзора, которая может работать за счёт разного количества камер и позволяет получать довольно точные и детализированные данные о пространственном окружении транспортного средства без необходимости сложной и дорогостоящей во всех отношениях обработки трёхмерных облаков точек с лидара.

Однако при таком подходе ключевой проблемой является *сегментация* полученных данных и выделение на них значимых объектов, таких как препятствия, люди, дорожные знаки с разметками и зоны, пригодные для передвижения. В данном случае, без надёжного и устойчивого алгоритма сегментации невозможно построение корректной сегментированной локальной карты на основе системы двухмерного кругового обзора, что, в свою очередь, затрудняет эффективное, а что самое главное безопасное планирование маршрута движения беспилотных наземных транспортных средств.

Актуальность этого исследования обусловлена растущими требованиями к автономным транспортным системам, нуждающимся в достаточно точных и вычислительно эффективных алгоритмах восприятия окружающей их среды для повышения безопасности и эффективности транспортной инфраструктуры подобного рода.

1 Постановка задачи и анализ предметной области

1.1 Проблема

Если представить себе среднестатистическое наземное транспортное средство, например, автомобиль, то на ум скорее всего сразу придёт относительно габаритное (до нескольких метров в длину и ширину) техническое устройство, движение которого хотя и довольно предсказуемо (держим руль ровно – едем прямо), но обычно сопровождается некоторой относительно высокой скоростью (речь как правило идёт о десятках метров в секунду) и инерцией, которая усложняет задачу быстрого изменения этой самой скорости в какой-нибудь чрезвычайной ситуации, например, когда автомобилю нужно резко остановиться перед внезапно появившимся на дороге препятствием в виде человека, животного, другого автомобиля и тому подобного. Скорость реакции водителя, оказавшегося в подобной ситуации, будет измеряться в десятых долях секунды, а теперь представим, что за управление отвечает не человек, а компьютер, и за рулём движущегося на высокой скорости и инертного автомобиля вообще никого нет, то есть, наше наземное транспортное средство – беспилотное.

Из этого длинного предисловия вытекает сразу несколько фундаментальных проблем данного вида транспорта, а именно: 1) **как заставить компьютер столь же точно и надёжно**, как это само собой получается у человека-водителя, **воспринимать** зачастую неоднозначную и комплексную **окружающую среду**; 2) **своевременно**, чуть ли не моментально, **обновлять эту информацию, всегда поддерживая её в самом актуальном состоянии**, и 3) на её основе **принимать взвешенные, прозрачные и понятные** для самих разработчиков подобных систем **решения**, причём делать это, как было выяснено выше, за **десятие доли секунды**, а то и быстрее, **при этом стабильно отрабатывая в 100% непохожих друг на друга случаев**.

В данной работе предпринимается попытка к частичному решению описанных выше проблем именно с точки зрения того самого инженера по восприятию и навигации беспилотных наземных транспортных средств, цель которого – научить компьютер управлять автомобилем так же, как это может делать он (разработчик) сам.

Кроме этого, можно выделить несколько более конкретных проблем:

- 1. Технические ограничения сенсоров и двухмерность системы кругового обзора**, что не только ограничивает дальность обзора и, как следствие, область видимости на построенной локальной карте, но и заставляет задействовать дополнительные алгоритмы и датчики для восстановления информации из третьего измерения;
- 2. Отсутствие семантической информации в данных по умолчанию**, что может затруднять принятие решений на этапе планирования и построения маршрута движения;
- 3. Отсутствие единых стандартов в области и закрытость частных решений**, в том числе формата и алгоритмов сегментации, что усложняет переносимость решений между различными платформами и роботами, а также существенно замедляет скорость разработки.

1.2 Цели и задачи

Целью данной работы является изучение современных подходов к разработке систем восприятия БНТС окружающего пространства и создание на их основе программного решения, позволяющего тестировать алгоритмы картирования и навигации транспортного средства по построенным картам в виртуальной среде.

Для достижения поставленной цели решаются следующие задачи:

1. Анализ существующих подходов к восприятию БНТС окружающего пространства и их применения при планировании маршрута движения;

2. Подготовка виртуальной среды и правильная настройка (калибровка) используемых в ней сенсоров;
3. Разработка системы двухмерного кругового обзора;
4. Построение на её основе сегментированной локальной карты;
5. Объединение готовых алгоритмов планирования маршрута движения транспорта с форматом построенной карты;
6. Тестирование и анализ эффективности предложенного подхода в симуляционной среде.

1.3 Обобщённая постановка задачи

После определения основной проблемы, а также целей и задач для её решения, обобщённое техническое задание будет выглядеть следующим образом:

Требуется разработать метод построения сегментированной локальной карты для планирования маршрута движения беспилотных наземных транспортных средств на основе системы двухмерного кругового обзора.

Разрабатываемый метод должен обеспечивать приемлемое качество выходных данных, получаемых от системы двухмерного кругового обзора, их точную сегментацию и построение итоговой локальной карты, пригодной для безопасной навигации транспортного средства.

В целях реализации данного метода будут использованы различные алгоритмы компьютерного зрения из библиотеки OpenCV и методы глубокого обучения через применение таких моделей свёрточных нейронных сетей, как *YOLO* и *FastSeg*, а также некоторые из подходов к построению не только локальной, но и глобальной карты совместно с динамическим планированием по ней маршрута движения БНТС. Сами карты будут включать в себя классификацию областей на проходимые и непроходимые зоны, а также учитывать статические и, возможно, динамические препятствия.

Поскольку архитектура всего решения будет написана на языке программирования Python и с использованием такого фреймворка для написания программ в области робототехники, как *ROS 2*, то для построения глобальной карты и последующего осуществления навигации по ней, будут задействованы такие готовые библиотеки, написанные специально под ROS 2, как SLAM Toolbox и Nav2 соответственно.

Для оценки эффективности разработанного метода, будет проведено его ручное тестирование в робототехническом 3D-симуляторе Webots, с визуальным анализом точности сегментации и классификации различных объектов на итоговых картах, а также общего качества построения маршрута.

Предполагается, что разрабатываемая автоматизированная навигационная система будет в том или ином её виде использоваться и дальше развиваться в проектах других разработчиков беспилотных транспортных средств (преимущественно наземных) в качестве вспомогательного модуля восприятия, который позволит им либо полностью отказаться от использования такого дорогостоящего и тяжёлого в вычислительном плане сенсора, как лидар, либо дополнять его данные информацией с совершенно различных типов видеокамер, которые при этом гарантированно будут покрывать область обзора в 360° вокруг транспортного средства против 270° , а зачастую и ещё меньше, у твёрдотельных (статических) лидаров.

1.4 Расширенная постановка задачи

1.4.1 Потоки обработки данных

Входные данные:

- *RGBD-изображения* со стереокамер в формате байтов с последующим их преобразованием в NumPy-массивы или OpenCV-матрицы;
- Облако точек с 3D-лидара в формате PointCloud2 с последующим его преобразованием в LaserScan;

- Глобальные координаты *эго-автомобиля* в формате метров относительно начальной точки мира в робототехническом симуляторе Webots;
- Крен (англ. roll), тангаж (англ. pitch), рыскание (курс, англ. yaw) *эго-автомобиля* как в формате углов Эйлера, так и в формате кватерниона, в зависимости от области применения.

Этапы обработки:

1. Получение данных с сенсоров;
2. Настройка и калибровка видеокамер;
3. Организация системы двухмерного кругового обзора;
4. Построение на её основе локальной карты;
5. Сегментация построенной карты;
6. Построение глобальной карты;
7. Её интеграция в модуль навигации.

Выходные данные:

- Сегментированная локальная карта в виде единого «сшитого» на основе системы двухмерного кругового обзора изображения в формате NumPy-массива;
- Глобальная карта в формате Occupancy Grid (сетка 2D-пространства, где каждой ячейке присваивается вероятность её занятости препятствием);
- Маршрут движения беспилотного наземного транспортного средства в формате точек траектории, построенный с учётом актуальной глобальной карты и направленный на достижение целевой точки;
- Управляющие команды в формате [*<скорость>* (км/ч), *<угол поворота колёс>* (градусы)].

Ограничения и допущения:

- Учёт только двухмерных данных, что может повлиять на полноту восприятия БНТС трёхмерного окружающего пространства;
- Около идеальные условия работы сенсоров и отсутствие сильных помех при их совместной работе;
- Предсказуемое и контролируемое распределение объектов на виртуальной сцене, световые и погодные условия которой постоянны и неизменны.

1.4.2 Функциональные требования

Решение должно реализовывать следующие функциональные возможности:

1. Получение данных с виртуальных стереокамер, 3D-лидара, GPS- и IMU-модулей из Webots в ROS 2 в их стандартных форматах (например, PointCloud2 для лидара);
2. При необходимости, преобразование этих данных в смежные системы координат или форматы (например, из углов Эйлера в кватернион для инерциальной системы);
3. Настройка и калибровка эмулируемой модели реальной видеокамеры со схожими характеристиками и возможностью масштабирования количества одновременно задействуемых устройств;
4. Организация системы двухмерного кругового обзора на основе объединения пяти выровненных изображений, поступающих с круговых камер в режиме реального времени;
5. Построение на её основе локальной карты в формате единого изображения с наложением на него информации от прочих сенсоров (например, точки маршрута движения от системы глобального позиционирования);

6. Выделение на построенной локальной карте областей пространства по их принадлежности к определённым классам (например, «дорога» и «не дорога») путём применения такого метода глубокого обучения как сегментация через обучение и применение свёрточных нейронных сетей на основе регионов (англ. Region Based Convolutional Neural Networks);
7. Построение глобальной карты на основе данных с лидара и локализация на ней этого-автомобиля (определение его месторасположения) за счёт данных одометрии (информация о его перемещении в пространстве с течением времени);
8. Преобразование формата построенной глобальной карты из Occupancy Grid в схожий с ним Costmap, который пригоден для использования готовых алгоритмов планирования маршрута движения беспилотных наземных транспортных средств с динамическим пересчётом траектории при появлении новых препятствий;
9. Визуальное представление данных с сенсоров, карт, текущего положения и ориентации робота, траектории его движения и прочей полезной для отладки и сбора статистики информации в специально предназначеннной для этого утилите под названием «RViz».

1.4.3 Нефункциональные требования

Решение должно удовлетворять следующим нефункциональным требованиям:

1. Производительность. В зависимости от типа сенсора, система должна обрабатывать их данные с частотой от 10 до 256 Гц. Полный цикл обновления сегментированной локальной карты, перестройки маршрута движения при необходимости и формирования последующей управляющей команды должен укладываться в 100-200 мс;

2. Надёжность. Система должна быть устойчива к частичным пропускам данных и временным помехам (например, переполнение очереди сообщений с лидарными сканами). При критических ошибках должна немедленно инициироваться безопасная остановка транспортного средства с возможностью корректного возобновления работы алгоритмов с момента остановки ТС и продолжения его движения;
3. Совместимость. Программный модуль должен быть совместим с популярными и признанными в области робототехники платформами для симуляции и автономного управления;
4. Модульность. Архитектура системы должна быть модульной для возможности быстрой и удобной замены, а также тестирования отдельных её компонентов (моделей и количества используемых датчиков, подходов к сегментации, планировщика для навигации по построенным картам и т. д.);
5. Масштабируемость. Как следствие двух предыдущих пунктов, решение должно быть адаптируемо под различные размеры и типы транспортных платформ, а также качество, плотность и количество сенсорных данных;

1.5 Анализ предметной области

Одной из основных задач отрасли беспилотного транспорта является необходимость создания эффективных программных решений, обеспечивающих надёжное и безопасное передвижение автономных транспортных средств в реальных условиях. Современные БНТС требуют высокоточных систем восприятия окружающей среды для такого рода перемещений. Основным элементом таких систем является построение карт окружающего транспортное средство пространства, которые используются для планирования его маршрута движения и следования по нему. В этом контексте сегментированные локальные карты играют важную роль, позволяя в первую очередь выделять препятствия и свободные области (условную дорогу), находящиеся в непосредственной близости (до 20-ти метров) от БНТС, а также учитывать прочие статические условия окружающей среды.

Один из подходов к формированию такого рода карт основан на использовании систем двухмерного кругового обзора, которые, исходя из названия, позволяют отслеживать всё, что происходит вокруг робота на цельные 360 градусов обзора с так называемого вида с высоты птичьего полёта (англ. Bird's-Eye view, далее – *BEV*), получаемого за счёт применения особого вида геометрических преобразований, а также перекрытия и последующего объединения изображений с нескольких видеокамер, установленных по периметру БНТС.

1.5.1 Анализ существующих подходов

Прежде, чем приступить к выбору используемых технологий и инструментов, с последующими этапами проектирования и реализации решения, совершенно не лишним будет провести анализ существующих подходов к решению обозначенных выше проблем и задач, а также к достижению поставленной цели и соответствуанию основным требованиям.

Поскольку минимальный набор для реализации системы двухмерного кругового обзора, построения на её основе локальной карты с применением к ней сегментации и планирования маршрута движения по построенной карте включает в себя лишь возможность получения изображений с видеокамер и дальнейшее их использование в алгоритмах компьютерного зрения, написанных на любых из наиболее популярных языках программирования, а также возможность удобной настройки и наглядной визуализации всего этого процесса в каком-нибудь симуляторе, то можно определить следующий набор критериев для сравнительного анализа:

1. Наличие симулятора с поддержкой как 2D-, так и 3D-сенсоров, готовых моделей транспортных средств и возможностью настройки их параметров, а также получения данных программным путём (например, через связь симулятора с отдельно написанной программой на языке Python или C++);
2. Наличие готовых инструментов, использование которых не только увеличит скорость разработки решения, но и позволит расширить его функциональные возможности, например, за счёт применения современных алгоритмов и подходов к построению карт и навигации, а также дополнительных утилит для визуализации (помимо самого симулятора);
3. Высокий уровень производительности и совместимости даже на самом мощном железе.

Apollo – платформа с открытым исходным кодом для создания программного обеспечения в области автономного вождения, разработанная компанией Baidu [4]. Содержит полный стек для автономного управления: восприятие, локализацию, планирование, управление, мониторинг, визуализацию и карты высокой точности. Кроме этого, Apollo предоставляет мощную симуляционную среду Dreamview и поддерживает интеграцию с другими симуляторами, такими как например SVL (описан ниже).

Autoware – набор программных компонентов с открытым исходным кодом, который построен на базе Robot Operating System (далее – ROS, описан ниже). Он предназначен для использования как в автономных транспортных системах, таких как автомобили, так и в других типах беспилотных транспортных средств [5].

Основные возможности Autoware включают в себя распознавание объектов, планирование и управление маршрутом движения, а также взаимодействие с окружающей средой посредством камер, лидаров и других типов датчиков.

CARLA (Car Learning to Act) – симулятор для автономного вождения и алгоритмов машинного обучения, предоставляющий подробные трёхмерные городские среды для исследования и разработки [6].

В CARLA можно моделировать не только различные типы транспортных средств, такие как легковые автомобили, грузовики, автобусы и велосипеды, но и дорожное движение, пешеходов, погодные условия (дождь, снег, туман) и различные сценарии взаимодействия на дороге.

Симулятор обеспечивает широкие возможности для настройки среды, включая изменение топографии, создание различных типов дорог и жилых районов, а также применение различных сценариев вождения и тестирования. Кроме того, Car Learning to Act позволяет встраивать в симуляцию различные алгоритмы и модели машинного обучения для проверки их работы в реалистичных условиях.

Isaac Sim – высокопроизводительная симуляционная платформа для создания и тестирования алгоритмов робототехники и автономного вождения, ориентированная на разработку с использованием GPU и нейросетевых вычислений [7]. Isaac Sim основан на графическом движке Omniverse, который обеспечивает реалистичную физику, фотореалистичную графику, поддержку синтетических данных и глубокой интеграции с библиотеками компьютерного зрения и искусственного интеллекта.

ROS + Webots/Gazebo – фреймворк с открытым исходным кодом, предназначенный для создания робототехнических приложений. Представляет из себя целую экосистему инструментов, библиотек и пакетов для картирования, навигации, визуализации, управления и взаимодействия с различными типами сенсоров. Всё это, несомненно, упрощает разработку сложных и модульных систем, однако, из-за отсутствия встроенной поддержки реального времени и распределённых систем, первая версия данного фреймворка имеет ограничения при промышленном использовании.

Вторая версия Robot Operating System пытается решить архитектурные ограничения предыдущей версии за счёт своего построения вокруг службы распространения данных ([англ.](#) Data Distribution Service, далее – *DDS*), что обеспечивает надёжную многопоточную и распределённую коммуникацию между узлами, встроенную поддержку реального времени, отказоустойчивости и безопасности.

Webots – кроссплатформенный 3D-симулятор с открытым исходным кодом для моделирования и программирования мобильных роботов. Предоставляет встроенную физику, поддержку разнообразных сенсоров (как 2D, так и 3D), готовые модели роботов и транспортных средств, а также возможность написания контроллеров для них на таких языках программирования как: Python, C++, Java и MATLAB. Активно используется в образовательных и исследовательских целях, обладает интуитивно понятным интерфейсом и, что самое главное, поддерживает интеграцию с ROS 1 и ROS 2, включая двустороннюю передачу данных.

Gazebo – мощный робототехнический симулятор с поддержкой реалистичной физики, 3D-визуализации и моделирования сложных сенсорных систем [8]. Широко используется в академических и промышленных проектах, особенно в связке с ROS.

Кроме этого, Gazebo предоставляет гибкие средства настройки симуляции, поддержку плагинов, моделирование динамики и взаимодействий с окружающей средой.

Оба симулятора, как Webots, так и Gazebo, поддерживают *SLAM*, навигационные и прочие стеки, а также позволяют запускать пользовательские ROS-пакеты.

SVL Simulator – мощный симулятор с открытым исходным кодом для автономного вождения, ранее известный как LGSVL [9]. Обеспечивает реалистичное моделирование городских и шоссе-сценариев.

Поддерживает мультисенсорные конфигурации, включая камеры, лидары, инерциальные измерительные модули (IMU) и системы глобального позиционирования (GPS).

С учётом определённого набора критериев и вышеперечисленного описания наиболее популярных и признанных в области робототехники платформ для симуляции и автономного управления, была составлена следующая сравнительная таблица:

Таблица 1. Сравнение существующих подходов

Критерий	Apollo	Autoware	CARLA	Isaac Sim	ROS 2 + Webots	ROS 2 + Gazebo	SVL Simulator
Симуляция сенсоров	Сторонние плагины	Зависит от симулятора	Полная	Полная	Полная	Полная	Полная
Готовые модели ТС	+	+	+	+	+	+	+
Настройка параметров	Конфиги	Ограничена	Высокая гибкость	Высокая гибкость	Удобная	Гибкая	Гибкая
Python/C++ API	+	+	+	+	+	+	+
Готовые инструменты	Внешние интеграции	Через ROS	Ручная интеграция	Встроенные плагины	Экосистема ROS	Экосистема ROS	Совместим с Autoware
Визуализация	Dreamview	RViz, Dreamview	RViz, Foxglove, PlotJuggler	Omniverse Kit UI	RViz, Foxglove, PlotJuggler	RViz, Foxglove, PlotJuggler	RViz, Web UI, сторонние GUI
Требования к железу	Высокие	Средние	Высокие	Высокие	Низкие / Средние	Средние / Высокие	Средние / Высокие
Простота	Низкая	Средняя	Низкая	Средняя	Высокая	Средняя	Низкая

Таким образом, аналогов разрабатываемого решения в свободном доступе в сети Интернет и с открытым исходным кодом найти не удалось, однако, как это было отмечено выше, «поскольку минимальный набор для реализации системы двухмерного кругового обзора, построения на её основе локальной карты с применением к ней сегментации и планирования маршрута движения по построенной карте включает в себя лишь возможность получения изображений с видеокамер и дальнейшее их использование в алгоритмах компьютерного зрения, написанных на любых из наиболее популярных языках программирования, а также возможность удобной настройки и наглядной визуализации всего этого процесса в каком-нибудь симуляторе», то проведённый сравнительный анализ существующих к этому подходов, по набору определённых там же критериев, показал (см. Табл. 1 выше), что связка фреймворка Robot Operating System совместно с робототехническим 3D-симулятором Webots является для этого наиболее предпочтительной.

1.5.2 Описание используемых технологий и инструментов

- **Python** – это высокоуровневый язык программирования с динамической типизацией и автоматическим управлением памятью, обладающий широкой экосистемой библиотек и удобным синтаксисом [10]. Активно используется в научных и инженерных задачах, в том числе в робототехнике, машинном обучении, компьютерном зрении и разработке прототипов. Благодаря множеству готовых библиотек (например, NumPy, OpenCV, PyTorch, ROS 2 API и другие), Python обеспечивает быструю реализацию и тестирование алгоритмов обработки данных с сенсоров, а также взаимодействие между различными модулями системы;
- **OpenCV** – это одна из самых популярных библиотек компьютерного зрения, включающая широкий набор инструментов для обработки изображений и видео, распознавания объектов, трекинга, калибровки различных типов камер и так далее [11].

В контексте данного проекта, OpenCV будет использоваться для калибровки виртуальных видеокамер, а также на протяжении всего этапа организации и работы системы двухмерного кругового обзора;

- **YOLO** – это семейство моделей сетей глубокого обучения для *детекции* объектов на изображениях и видео в режиме реального времени [12]. Основной принцип работы заключается в однократном проходе изображения через свёрточную нейронную сеть, что обеспечивает высокую скорость обнаружения объектов при приемлемой точности.

В контексте данного проекта, YOLO будет использоваться для распознавания препятствий вокруг этого-автомобиля, что позволит учитывать их при сегментации локальной карты и планировании маршрута движения;

- **FastSeg** – это легковесная модель сегментации, предназначенная для быстрого выделения объектов на изображениях [13]. Основанная на архитектуре типа encoder-decoder, FastSeg обеспечивает высокую скорость инференса (предсказания) как на CPU, так и на GPU даже в условиях ограниченных вычислительных ресурсов.

В контексте данного проекта, FastSeg будет использоваться для выделения проходимых и непроходимых областей на изображениях с камер кругового обзора, что является ключевым шагом при построении сегментированной локальной карты;

- **ROS 2** – это современная версия фреймворка ROS, обеспечивающего модульную и распределённую архитектуру программ для робототехники [14]. ROS 2 реализует взаимодействие между различными компонентами системы через публикацию и подписку на сообщения из так называемых топиков ([англ. topics](#)), вызовы сервисов ([англ. services](#)) и действия ([англ. actions](#)).

Предлагает улучшенную поддержку реального времени, многопоточности, надёжности и безопасности, а также совместим с современными DDS-стандартами.

В контексте данного проекта, Robot Operating System будет использоваться для организации связей между сенсорами из Webots и алгоритмами обработки данных, поступающих с них;

- **Webots** – это кроссплатформенный 3D-симулятор с открытым исходным кодом для моделирования и программирования мобильных роботов [15], как уже было отмечено выше. Он поддерживает моделирование как 2D-, так и 3D-сенсоров, позволяет конфигурировать окружение, многообразные модели роботов и транспортных средств, а также управлять симуляцией с помощью кода на Python, C++, Java и MATLAB.

В контексте данного проекта, Webots будет использоваться как основная симуляционная среда, в которой моделируется поведение БНТС, а также осуществляется сбор данных с виртуальных сенсоров;

- **SLAM Toolbox** – это библиотека для ROS 2, предназначенная для одновременного построения глобальной карты и определения месторасположения (локализации) робота на ней (расшифровка аббревиатуры SLAM) [16].

В контексте данного проекта, SLAM Toolbox будет использоваться для построения глобальной карты, локализации и навигации на ней;

- **Nav2 + AMCL** – это фреймворк в ROS 2, предназначенный для автономной навигации и обеспечивающий уже упомянутую ранее локализацию, а также планирование и управление маршрутом движения автомобиля с объездом препятствий [17].

AMCL – это метод локализации, основанный на алгоритме частиц, и используемый, когда карта уже известна и требуется определить положение и вращение робота на ней. В отличие от SLAM Toolbox, AMCL не строит карту с нуля, а только определяет, где робот находится на уже существующей.

В контексте данного проекта, Nav2 и AMCL будут использоваться для непосредственного осуществления навигации и локализации на построенной библиотекой SLAM Toolbox глобальной карте, обеспечивая генерацию оптимального пути к цели с учётом статических и, возможно, динамических препятствий.

1.5.3 Обоснование выбора технологической платформы

В подразделе 1.5.1 «**Анализ существующих подходов**» выше, особое внимание было уделено выбору технологической платформы, которая обеспечивала бы:

1. Поддержку как 2D-, так и 3D-сенсоров;
2. Наличие готовых моделей наземных транспортных средств с гибкой настройкой параметров как их самих в частности, так и всей симуляционной среды в целом;
3. Возможность интеграции с внешними программными компонентами, написанными на различных языках программирования;
4. Наличие готовых инструментов для картирования, навигации, визуализации и отладки решения;
5. Высокий уровень производительности и совместимости даже на самом мощном железе.

На основе этих критериев были рассмотрены и проанализированы несколько наиболее популярных платформ в области робототехники и автономных транспортных систем, в результате чего наиболее подходящей из них, для решения поставленных проблем, был выбран стек ROS 2 в связке с Webots из-за его следующих преимуществ перед конкурентами:

1. Webots предоставляет нативную поддержку 2D- и 3D-сенсоров, таких как: RGB- и RGBD-камеры, лидары, IMU, GPS, акселлерометры, гироскопы, компас и многих других с возможностью средней по гибкости настройкой параметров и имитацией их физического поведения, а также множество готовых моделей роботов: от пылесосов и собак до роборук и автомобилей;
2. Благодаря ROS-интерфейсу и API, возможно напрямую взаимодействовать с симулятором через Python, C++, Java или MATLAB, что критично при использовании таких современных библиотек компьютерного зрения и глубокого обучения, как: OpenCV, YOLO, FastSeg или любой другой модели нейронной сети, функционал которой, например, сильно завязан на том же PyTorch;
3. Webots имеет официальную интеграцию с ROS 2, что обеспечивает лёгкое подключение и взаимодействие с такими готовыми инструментами, как:
 - SLAM Toolbox и AMCL для картирования и локализации;
 - Nav2 для навигации;
 - RViz для визуализации.
4. В отличие от более ресурсоёмких платформ (Apollo, CARLA, Isaac Sim), Webots может стабильно работать на оборудовании со средними характеристиками, не требуя при этом наличия мощной видеокарты.

Таким образом, выбор технологической платформы в лице Webots и ROS 2 обоснован балансом между простотой, но при этом гибкостью использования, поддержкой необходимых инструментов и производительностью, что делает её оптимальным выбором для реализации поставленных в данной работе целей и задач, а также для решения обозначенных проблем в условиях, приближенных к реальным.

2 Проектирование и реализация решения

Переход от теоретических предпосылок к практической реализации требует тщательного проектирования архитектуры системы, её основных модулей, составляющих их компонентов, а также взаимосвязей между ними.

Данная глава нацелена на формирование этой основы будущего программного решения, поскольку этап его проектирования выступает как бы связующим звеном между теоретической моделью системы и её реализацией в виде работающей программы, интегрируемой впоследствии в ещё более общую архитектуру всего беспилотного наземного транспортного средства в целом.

2.1 Общая архитектура

На Рисунке 1 ниже приведена общая архитектура решения, которая представляет из себя набор модулей, связей между ними в виде типа пересылаемых сообщений, а также сторонних подключаемых библиотек, своих для каждого из модулей:

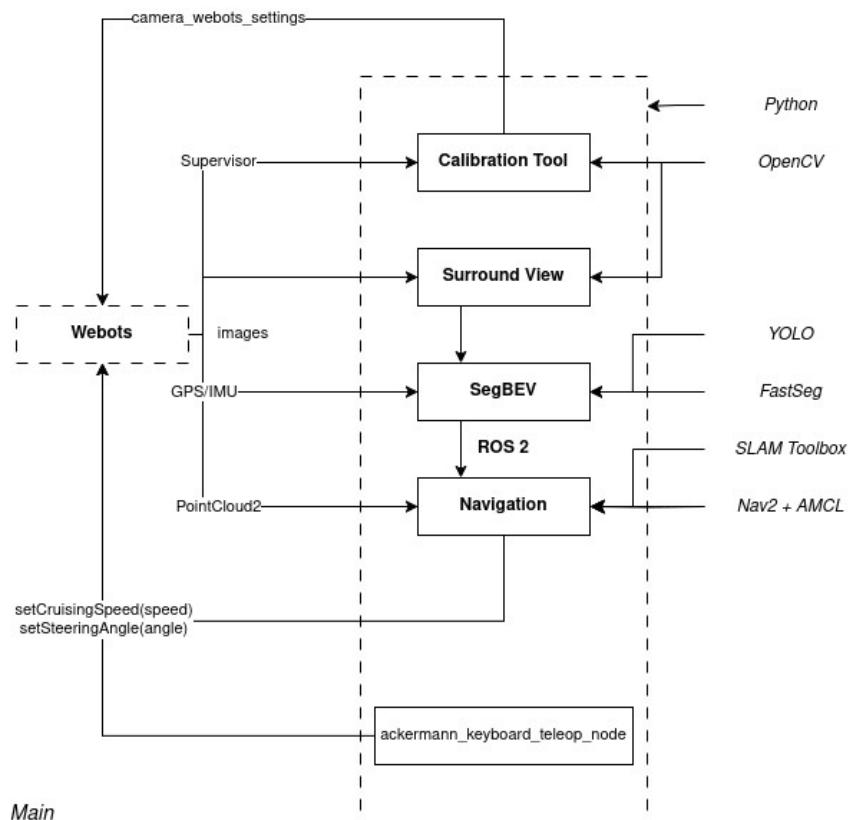


Рис. 1. Архитектура всего решения

Робототехнический 3D-симулятор Webots взаимодействует с фреймворком для робототехники ROS 2 посредством основного используемого языка программирования Python и системы топиков, которые служат для передачи в и изъятии из них данных с сенсоров; системы узлов, которые обрабатывают эти данные, а также системы контроллеров, которые позволяют отправлять команды управления (скорость и угол поворота колёс) из фреймворка в симулятор, непосредственно на автомобиль. В режиме автономного управления такие управляющие команды отправляет модуль «Navigation», путём вызова методов setCruisingSpeed и setSteeringAngle у объекта робота. В режиме ручного управления, это делается отдельным узлом под названием «ackermann_keyboard_teleop_node», который позволяет управлять БНТС с клавиатуры.

Другая очевидная функция, которую выполняет Webots – это визуализация объекта этого-автомобиля, а также пространства, которое его окружает (лес, дорога, препятствия и т. п.). Более наглядно, работа этого робототехнического симулятора представлена в главе 3 «**«Тестирование решения»**.

Модуль «**Calibration Tool**» – позволяет через конфиг camera_webots_settings настроить конкретную реальную модель видеокамеры, которую планируется использовать (см. экс. №1 в подразделе 3.1.1), после чего откалибровать её с помощью библиотеки OpenCV, специального класса Supervisor и поступающих с камер изображений images, как это описано в следующем разделе 2.2 «**«Калибровка виртуальных видеокамер»**.

Модуль «**Surround View**» – система двухмерного кругового обзора, которая по всё тем же images и с использованием OpenCV формирует вид сверху на беспилотный автомобиль и то, что его окружает в симуляторе.

Более подробное описание того, как конкретно это происходит, представлено в разделе 2.3 «**«Система двухмерного кругового обзора»** текущей главы.

Модуль «**SegBEV**» – строит сегментированную локальную карту, как это описано в разделе 2.4 «**Сегментированная локальная карта**». Для этого он использует единое изображение кругового обзора – конечный продукт предыдущего модуля, а также две модели глубокого обучения – YOLO и FastSeg, которым на вход подаются цветные и выровненные изображения images.

GPS/IMU (см. Рис. 1) – это данные с одноимённых сенсоров, которые нужны для реализации частично автономного движения транспортного средства по глобальным спутниковым координатам с учётом его текущего курса. В дальнейшем всё это может использоваться для реализации уже полностью автономного движения, принимая во внимание информацию с локальной карты.

Конечный продукт данного модуля – единое сегментированное изображение кругового обзора, которое может подаваться на вход следующему модулю, дополняя и уточняя таким образом информацию, предназначенную для навигации БНТС с учётом и объездом препятствий.

Модуль «**Navigation**» – отвечает за полностью автономное движение это-автомобиля, принимая во внимание информацию с глобальной карты и используя для этого такие библиотеки как SLAM Toolbox и Nav2 + AMCL, описание которых было представлено выше, в подразделе 1.5.2 «**Описание используемых технологий и инструментов**» главы 1 «**Постановка задачи и анализ предметной области**».

На вход им подаётся LaserScan, полученный из PointCloud2, а вот что всё это такое и как конкретно осуществляется данный процесс подробно описано в разделе 2.5 «**Глобальная карта и планирование маршрута движения беспилотного наземного транспортного средства**» главы 2.

2.2 Калибровка виртуальных видеокамер

Поскольку система двухмерного кругового обзора, устройство которой будет более подробно описано в следующем разделе, полностью завязана на данные, поступающие от пяти отдельных виртуальных стереокамер модели ZED 2, то правильно было бы, перед началом какой-либо обработки этих данных, устранить в них любые возможные искажения, которые, применительно как к электронным оптическим системам, каковыми и являются большинство современных камер, так и к их продукту – RGB-изображениями, могут иметь следующую природу:

1. Радиальная *дисторсия*, которая изгибает прямые линии на изображении от («подушкообразная») или к («бочкообразная») его центру и задаётся следующими формулами по модели Брауна-Конради [18]:

$$x_{distorted} = x \left(1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6 \right) \quad (1)$$

$$y_{distorted} = y \left(1 + k_1 * r^2 + k_2 * r^4 + k_3 * r^6 \right) \quad (2)$$

где: $x_{distorted}, y_{distorted}$ – координаты с учётом искажения; x, y – нормализованные координаты точки (в системе координат камеры, где $z = 1$); r – расстояние от оптического центра камеры (чем дальше точка от центра изображения, тем сильнее искажения); $k_1 - k_3$ – нелинейные коэффициенты радиальной дисторсии;

2. Тангенциальная дисторсия, которая наклоняет вертикальные и горизонтальные линии на изображении, искажая таким образом перспективу. Задаётся следующими формулами по модели Брауна-Конради:

$$x_{distorted} = x + [2 p_{1xy} + p_2(r^2 + 2x^2)] \quad (3)$$

$$y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2xy] \quad (4)$$

где: p_1, p_2 – нелинейные коэффициенты тангенциальной дисторсии;

3. Другие виды и типы искажений [19], которые ввиду идеальных условий и данных в симуляторе не будут эмулироваться в данной работе.

Специально для того, чтобы устраниТЬ все описанные выше искажения, был написан отдельный модуль под названием «Calibration Tool», архитектура которого представлена на следующей диаграмме:

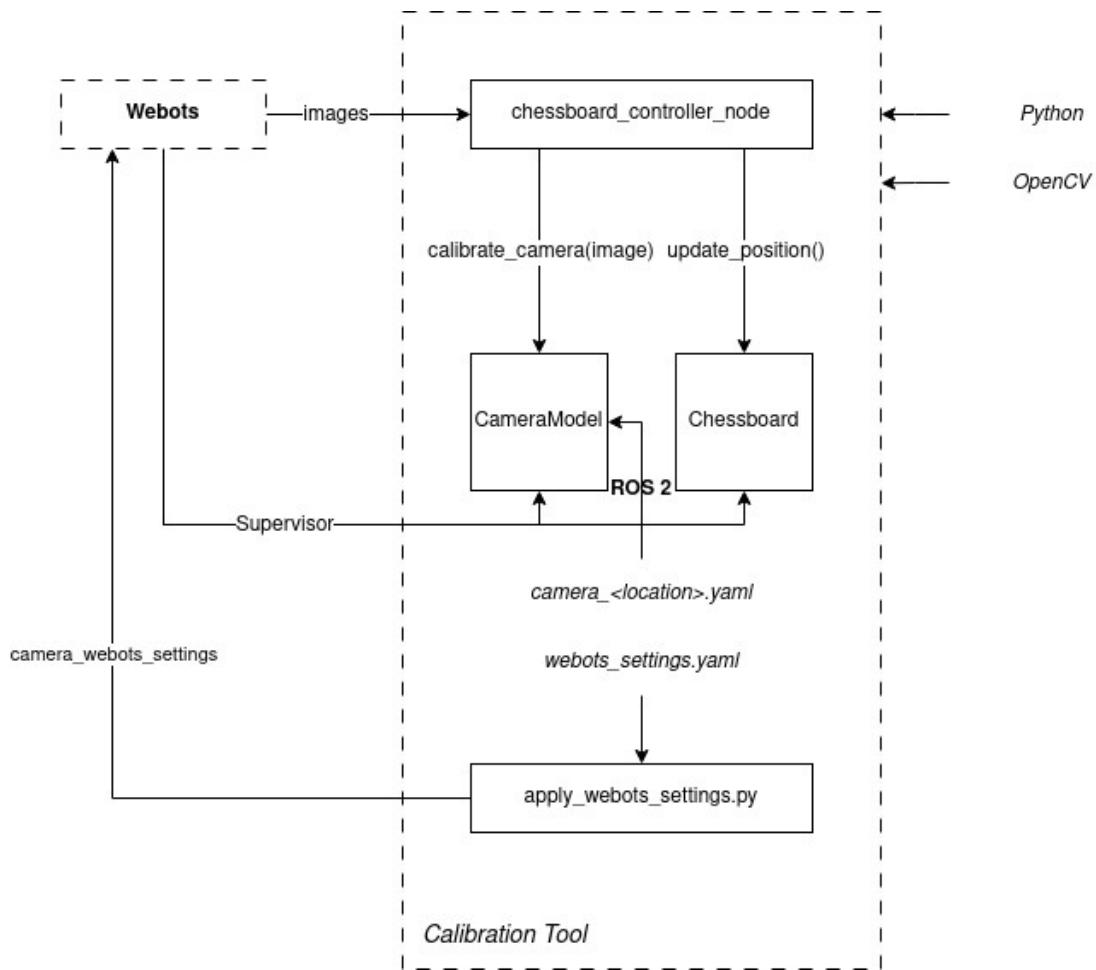


Рис. 2. Архитектура модуля «Calibration Tool»

Из робототехнического симулятора Webots, цветные искажённые изображения с размещённых в нём виртуальных неоткалиброванных видеокамер поступают на вход ROS-узла под названием «chessboard_controller_node», который передаёт их в качестве аргумента в метод `calibrate_camera` конкретного объекта той или иной используемой в симуляторе камеры (класс «`CameraModel`») для проведения дальнейшей её калибровки. Сразу же после этого, данный узел вызывает метод `update_position` у конкретного объекта той или иной используемой в симуляторе шахматной доски (класс «`Chessboard`»), расположенной напротив каждой из стереокамер, заставляя доску переместиться в следующие координаты из их заранее заданного списка траектории её движения.

Перемещать объекты «кодом» по сцене в Webots, а также изменять их параметры во время запуска симуляции позволяет так называемый Supervisor, который связывает компонент «`Chessboard`» с компонентом «`Webots`» (см. Рис. 2 выше).

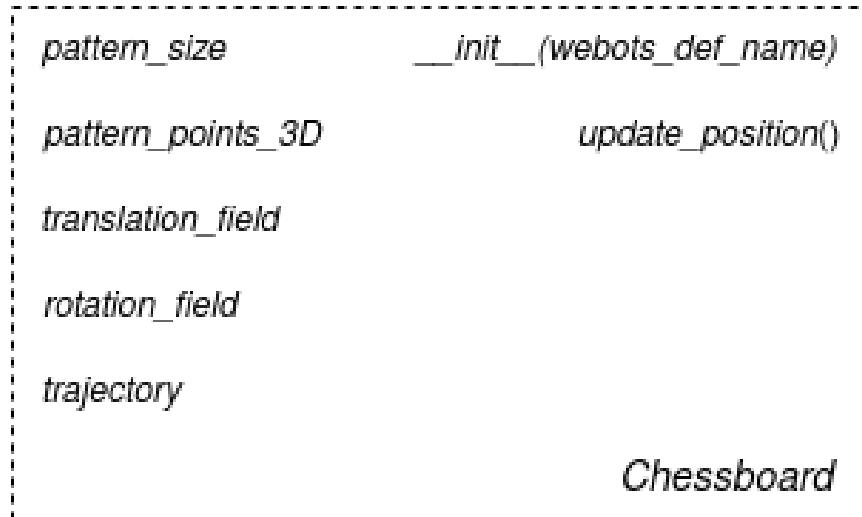


Рис. 3. Содержимое компонента «`Chessboard`»

1. **pattern_size** – задаёт размерность шахматной доски в метрах;
2. **pattern_points_3D** – список координат углов шахматной доски в трёхмерном пространстве. Зная размеры клеток доски в сантиметрах, можно конвертировать индексы точек пересечения паттерна в их реальные координаты X и Y. Координата по оси Z для каждой точки исключается, так как предполагается, что все они лежат в одной плоскости;
3. **__init__(webots_def_name)** – инициализирует объект шахматной доски с такими его параметрами, как: перемещение, вращение и траектория движения. Аргумент webots_def_name используется для идентификации конкретной доски в Webots по её *DEF-наименованию* с использованием Supervisor;
4. **update_position()** – обновляет месторасположение (перемещение и вращение) шахматной доски в симуляторе, учитывая заранее заданную траекторию её движения из словаря chessboards_movement_trajectory, который имеет следующую структуру:
 - Ключ – DEF-наименование доски. Например, 'chessboard_front_left';
 - Значение – многовложенный список координат в формате: ['ОСЬ', <значение>] для перемещения и ['ОСЬ', [<значение>, <значение>, <значение>, <значение>]] для вращения по кватерниону.

До тех пор, пока текущее значение заданной координаты не совпадает с требуемым хотя бы до одного знака после запятой, паттерн будет стремиться достичь его с заданной глобальной переменной CHESSBOARD_MOVEMENT_SENSITIVITY чувствительностью движения (0.1 метра или 10 сантиметров по умолчанию).

Конечное расположение шахматной доски на сцене обновляется через вызов встроенного в Webots метода `setSFVec3f` у **`translation_field`**, который задаёт новые значения по осям X, Y и Z, а конечная ориентация – через `setSFRotation` у **`rotation_field`** с передачей в него кватерниона (X, Y, Z и W) в качестве аргумента. По достижению определённой точки, она удаляется из списка **`trajectory`** вызовом метода `pop(0)` и процесс повторяется снова до тех пор, пока доской не будет пройдена вся заданная траектория её движения.

Одна из самых основных моделей всего решения под названием «`CameraModel`» представлена одноимённым компонентом, который имеет следующее содержимое:

<code>calibration_flags</code>	<code>__init__(webots_camera_name, load_parameters)</code>
<code>optical_characteristics</code>	<code>load_camera_parameters()</code>
<code>calibration_images_count</code>	<code>calibrate_camera(calibration_image)</code>
<code>chessboard</code>	<code>undistort(image)</code>
<code>image_shape</code>	<code>flip(image)</code>
<code>K</code>	<code>get_projection_matrix(image)</code>
<code>D</code>	
<code>projection_matrix</code>	
<code>rotation_vectors</code>	<code>object_points_3D</code>
<code>translation_vectors</code>	<code>image_points_2D</code>
<i>CameraModel</i>	

Рис. 4. Содержимое компонента «`CameraModel`»

1. **calibration_flags** – специальные флаги для более тонкой настройки процесса калибровки (см. Лист. 9);
2. **optical_characteristics** – заранее известные оптические характеристики определённой модели видеокамеры, которые позволяют приблизить симулируемую в Webots камеру к её реальному воплощению, как это было сделано при тестировании решения в эксперименте №1 раздела 3.1.1 главы 3.

Для избежания конфликтов в случаях, когда заданные значения горизонтального (horizontal_fov) и вертикального (vertical_fov) угловых разрешений не позволяют одновременно обеспечить заданных значений ширины (image_width) и высоты (image_height) изображений, поступающих с устройства, было принято решение ввести отдельный разграничительный параметр под названием «use_fov_values», зафиксировав при этом ширину изображения, а его высоту и фокусное расстояние видеокамеры в пикселях пересчитывать по следующим формулам:

$$imageHeight = imageWidth * \left(\frac{(\tan(verticalFOV/2))}{(\tan(horizontalFOV/2))} \right) \quad (5)$$

$$focalLength = \frac{imageWidth}{2 * \tan(horizontalFOV/2)} \quad (6)$$

3. **calibration_images_count** – количество калибровочных изображений (50 по умолчанию). Чем их больше, тем, в теории, точнее будут подобраны внутренние параметры камеры, однако, в пределах разумного, чтобы процесс калибровки не занимал слишком много времени и алгоритмы не пытались лучше подобрать и без того точные значения параметров;
4. **chessboard** – объект класса Chessboard (был подробно описан ранее), по данным от которого будет калиброваться та или иная видеокамера (заполняться недостающие данные у объекта класса CameraModel);

5. **image_shape** – пиксельное разрешение выходного изображения в формате [image_height, image_width, 4];
 6. **K** – матрица внутренних параметров, таких как фокусное расстояние и оптический центр;
 7. **D** – вектор коэффициентов дисторсий;
 8. **projection_matrix** – матрица проекции, которая преобразует исходное перспективное изображение в него же, но с видом сверху (см. Рис. 10);
 9. **rotation_vectors** – векторы (ось) вращения для каждого из калибровочных изображений, длина которых – это значение угла поворота камеры по одной из трёх осей относительно мировых координат. Задают ориентацию устройства в пространстве относительно сцены;
 10. **translation_vectors** – векторы перемещения для каждого из калибровочных изображений, длина которых – это значение смещения камеры по одной из трёх осей относительно центра мировых координат. Задают положение устройства в пространстве относительно сцены;
 11. **object_points_3D** – список обнаруженных углов шахматной доски в трёхмерном пространстве (реальный мир);
 12. **image_points_2D** – список обнаруженных углов шахматной доски в двухмерном пространстве (плоскость изображения);
- В данном решении, **rotation_vectors** и **translation_vectors** используются исключительно для расчёта описанной ранее ошибки перепроектирования, в то время как **object_points_3D** и **image_points_2D** также участвуют в данном процессе, но предназначены преимущественно не для этого, а для ключевого поиска сопоставлений обнаруженных алгоритмами библиотеки OpenCV углов шахматных досок из 3D-пространства реального мира в 2D-пространство изображения.

Непосредственный же расчёт ошибки (см. Лист. 10) производится следующим образом:

Пусть:

$X_j \in \mathbb{R}^3$ – 3D-координаты j-ой точки объекта;

$x_j^{obs} \in \mathbb{R}^2$ – наблюдаемая (реальная) 2D-точка на изображении;

$x_j^{proj} \in \mathbb{R}^2$ – проецированная 2D-точка, полученная через cv2.projectPoints.

Тогда ошибка перепроектирования для одного изображения (набора точек) вычисляется как:

$$e_i = \frac{1}{N_i} \sum_{j=1}^{N_i} \|x_j^{obs} - x_j^{proj}\|_2 \quad (7), \text{ где:}$$

N_i – количество точек на i-ом изображении;

$\|\cdot\|_2$ – евклидова норма.

Если у нас M изображений (или позиций объекта), то финальная средняя ошибка:

$$\text{Mean Reprojection Error} = \frac{1}{M} \sum_{i=1}^M e_i \quad (8)$$

Подставляя формулу выше:

$$\text{Mean Error} = \frac{1}{M} \sum_{i=1}^M \left(\frac{1}{N_i} \sum_{j=1}^{N_i} \|x_j^{obs} - x_j^{proj}\|_2 \right) \quad (9)$$

13. **__init__(webots_camera_name, load_parameters)** – инициализирует объект видеокамеры, аналогично объекту шахматной доски с использованием Supervisor. По умолчанию аргумент load_parameters имеет значение True;
14. **load_camera_parameters()** – загружает значения внутренних и внешних параметров камеры из конфигурационного .yaml-файла, – выходного результата вызова метода calibrate_camera –, и сохраняет их в соответствующие переменные, описанные выше.

Вызывается в конце метода `__init__` и выполняет подгрузку параметров только при том условии, если переменная `load_parameters` истинна, то есть, если устройство ранее уже калибровалось и его параметры известны;

15. **calibrate_camera(calibration_image)** – в начале своей работы распределяет оптические характеристики из словаря `optical_characteristics` по заданным переменным (определяет начальные значения внутренних и внешних параметров для их дальнейшего уточнения в процессе калибровки), после чего алгоритмы OpenCV пытаются найти на каждом `calibration_image` углы шахматной доски через вызов метода `cv2.findChessboardCorners`. Если им это удаётся, то максимальное количество требуемых калибровочных изображений `calibration_images_count` уменьшается на единицу, углы дополнительно уточняются через вызов метода `cv2.cornerSubPix` и добавляются в списки `object_points_3D` и `image_points_2D` из пунктов 11 и 12 выше соответственно. Процесс повторяется снова до тех пор, пока `calibration_images_count != 0`, после чего вызывается ключевой метод `cv2.calibrateCamera`, который подбирает конечные значения параметров и заменяет ими начальные из `optical_characteristics`, записывая их в выходной файл (см. Лист. 7). На этом процесс калибровки видеокамеры завершается.
16. **undistort(image)** – устраняет искажения на изображении `image`, перед этим дополнительно оптимизируя матрицу `K` и вектор `D` через вызов метода `cv2.getOptimalNewCameraMatrix`, а также рассчитывая по ним так называемые карты выравнивания ([англ. rectify map](#)) с помощью метода `cv2.initUndistortRectifyMap`, которые, по итогу, применяются для создания ровного изображения через вызов метода `cv2.remap`;

17. **flip(image)** – переворачивает `image` в зависимости от `self.device_name` – названия устройства, с которого это изображение пришло, а также в зависимости от его месторасположения относительно БНТС, если бы оно было направлено на условный магнитный север Земли и мы смотрели бы на транспортное средство сверху.

Таким образом, если передать в качестве аргумента для данного метода изображение с камеры, установленной по левую сторону автомобиля, оно будет повёрнуто им на 90 градусов против часовой стрелки и станет «смотреть» на условный запад; с камеры, установленной по правую сторону автомобиля, – на 90 градусов по часовой стрелке, а смотреть – на восток. Изображение с передней камеры останется без изменений, а с задней будет отзеркалено на 180 градусов относительно него и ориентировано на юг;

18. **get_projection_matrix(image)** – рассчитывает проекционную матрицу `projection_matrix` путём вызова метода `cv2.findHomography`, который принимает в качестве аргументов четыре точки `src` исходного перспективного изображения и четыре точки `dst` целевого изображения, которое требуется получить. Первые из них отмечаются вручную с использованием специально написанного для этого класса-утилиты под названием `PointSelector` (см. Рис. 10), в то время как `dst` берутся из файла с параметрами кругового обзора `bev_parameters.py` (см. Лист. 1) по всё тому же названию устройства `self.device_name`, и которые формируют из себя как бы конечный вид, в который требуется перевести точки из `src`, применив *гомографию*.

После того как матрица будет рассчитана, данный метод допишет её в конфигурационный файл `self.device_name.yaml`, который, помимо неё самой, также содержит в себе и другие параметры конкретной видеокамеры (см. Лист. 7).

```

near_shift_width = 200 # Расстояние в пикселях между областью автомобиля (см. ниже) и
near_shift_height = 200 # угловыми шахматными досками размерностью 6x5 (см. Рис. 26 ниже)

far_shift_width = 100 # Расстояние в пикселях за пределами угловых шахматных досок
far_shift_height = 100 # (чем больше эти значения, тем большую область покрывает круговой обзор)

total_width = 585 + 2 * far_shift_width # Итоговое разрешение единого сшитого изображения
total_height = 677 + 2 * far_shift_height #

# Координаты области изображения, занимаемой автомобилем:
# [(vehicle_leftside_edges_x, vehicle_topside_edges_y), (vehicle_rightside_edges_x, vehicle_bottomside_edges_y)]
vehicle_leftside_edges_x = far_shift_width + 50 + near_shift_width # Координата X левых углов
vehicle_rightside_edges_x = total_width - vehicle_leftside_edges_x # Координата X правых углов

vehicle_topside_edges_y = far_shift_height + 50 + near_shift_height # Координата Y верхних углов
vehicle_bottomside_edges_y = total_height - vehicle_topside_edges_y # Координата Y нижних углов

# Разрешение BEV-изображения для каждой из видеокамер (см. нижнее окно на Рис. 10 ниже)
projection_shapes = {

    ...
    'camera_front': (total_width, vehicle_topside_edges_y),
    ...
}

# Сохранённые координаты четырёх точек, отмечаемых на изображении с каждой из камер через
# инструмент PointSelector (см. верхнее окно на Рис. 10), строго в порядке их проставления
projection_src_points = {

    ...
    'camera_front': [
        (373, 410), # Верхняя левая
        (937, 410), # Верхняя правая
        (0, 531), # Нижняя левая
        (1304, 531), # Нижняя правая
    ],
    ...
}

# Координаты всех тех же четырёх точек выше в том же порядке, но образующие собой как бы фигуру
# прямоугольника – вид сверху на отмечаемую ранее через PointSelector жёлтую область (см. Рис. 10)
projection_dst_points = {

    ...
    'camera_front': [
        (far_shift_width + 109, far_shift_height),
        (far_shift_width + 470, far_shift_height),
        (far_shift_width + 109, far_shift_height + 155),
        (far_shift_width + 470, far_shift_height + 155),
    ],
    ...
}

```

Листинг 1. Параметры кругового обзора

Вызов метода cv2.warpPerspective применяет матрицу проекции к выровненному изображению, получая таким образом его вид сверху. С использованием немного другой математики, но всё той же матрицы, аналогичное преобразование проделывается и с ограничивающими рамками препятствий, которые удалось распознать одной из моделей глубокого обучения, более подробное описание которых будет представлено далее, в разделе 2.4 «**Сегментированная локальная карта**». Список с самими рамками может быть передан через необязательный аргумент рассматриваемого метода под названием «obstacle_bboxes», в то время как image (predicted_labels в 2.4 для наглядности) – это уже сегментированное перспективное изображение с одной из камер, каждый пиксель которого принадлежит к одному из классов («дорога» или «не дорога»), присвоенному ему другой моделью глубокого обучения.

В случае, если projection_matrix уже была рассчитана до этого и теперь стоит задача получить выровненное ([англ.](#) undistorted), преобразованное ([англ.](#) projected) и перевёрнутое ([англ.](#) flipped) изображение – конечный результат работы метода get_projection_matrix, то можно задействовать ещё один необязательный аргумент данного метода под названием «gotten», который позволит избежать пересчёта проекционной матрицы при каждом новом его вызове.

2.3 Система двухмерного кругового обзора

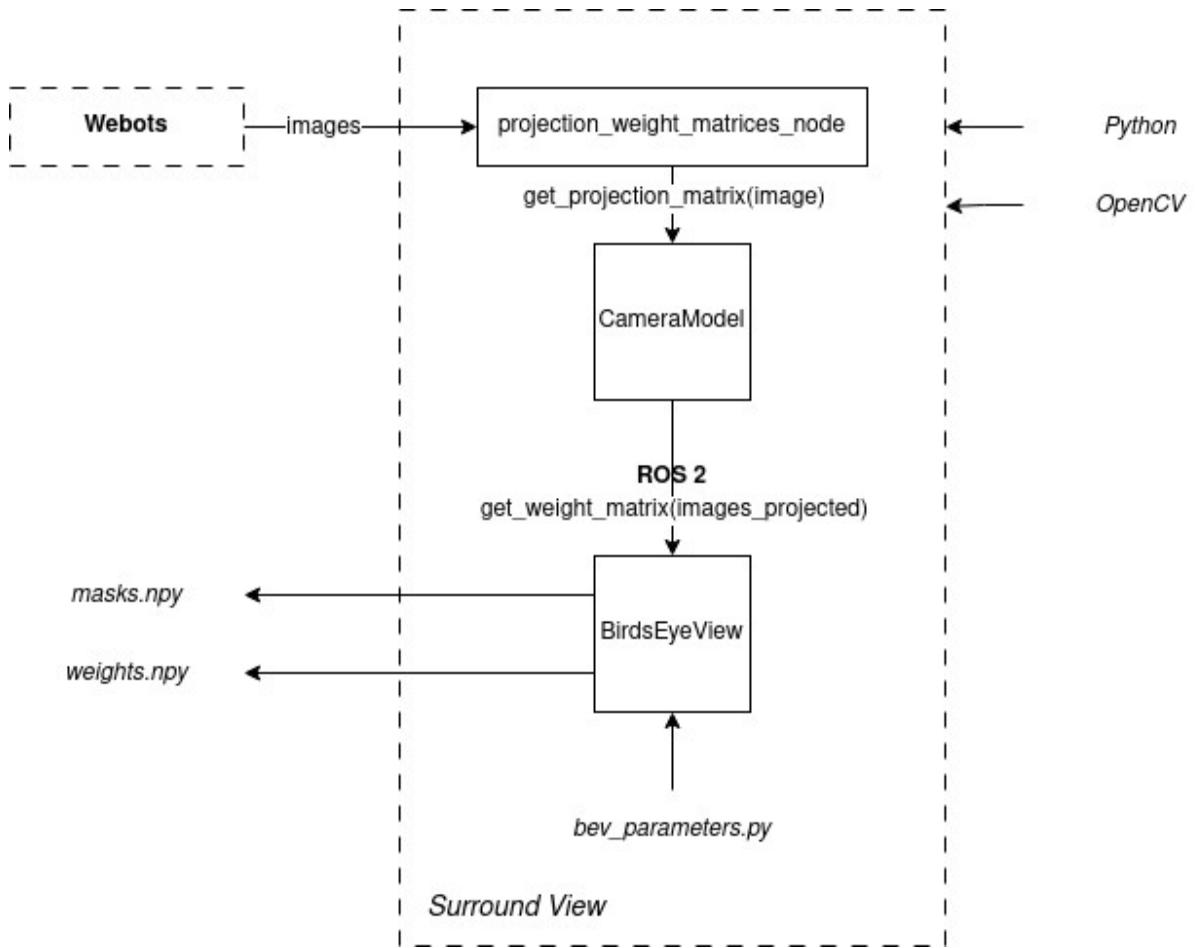


Рис. 5. Архитектура модуля «Surround View»

В узле «projection_weight_matrices_node» байтовые изображения `images`, поступающие с объектов видеокамер из Webots, преобразуются в формат NumPy-массивов и передаются так далее, в метод `get_projection_matrix`, логика работы которого была рассмотрена незадолго до этого.

Итоговое изображение для каждой из камер складируется в словарь `images_projected` с ключом в виде наименования `device_name` конкретного устройства, после чего, всё в том же узле, вызывается метод `get_weight_matrix`, который в качестве аргумента и принимает в себя этот словарь с выровненными, преобразованными и перевёрнутыми, в соответствии с месторасположением девайса, изображениями.

Внутри этого метода создаётся экземпляр класса-компоненты «BirdsEyeView» и поочерёдно вызываются его ключевые методы, описание которых, вместе с содержимым самого компонента, представлено ниже:

<i>xl, xr, yt, yb</i>	<i>init(images, load_weights_and_masks)</i>
<i>frames</i>	<i>load_weights_and_masks()</i>
<i>weights</i>	<i>get_weights_and_masks()</i>
<i>masks</i>	<i>luminance_balance()</i>
<i>image</i>	<i>merge(image_1, image_2, i)</i>
<i>@front_left</i>	<i>stitch()</i>
<i>...</i>	
<i>@right_central</i>	<i>white_balance()</i>
	<i>add_ego_vehicle_and_track_obstacles()</i>
	<i>get_upper_part(image)</i>
	<i>...</i>
	<i>fb_get_central_part(image)</i>
	<i>BirdsEyeView</i>

Рис. 6. Содержимое компонента «BirdsEyeView»

1. **xl, xr, yt, yb** – координаты левых, правых, верхних и нижних углов этого-автомобиля соответственно на итоговом едином изображении кругового обзора. Берутся из *bev_parameters.py* (см. Лист. 1 выше);
2. **frames** – список с изображениями из словаря *images_projected*, которые будут сшиваться, а также распознанные на их перспективных версиях ограничивающие рамки препятствий в виде синих пластиковых бочек, координаты их центра и расстояние до каждого из них в метрах, полученное из карт глубины (см. Рис. 7 ниже), по одной с каждой камеры;
3. **weights** – список весовых матриц *G*, полученных в процессе объединения *frames*;
4. **masks** – список масок *M*, полученных в процессе объединения *frames* и перекрывающихся областей на них.

Сам процесс и математика расчёта *G* и *M* подробно описаны ниже данного списка;

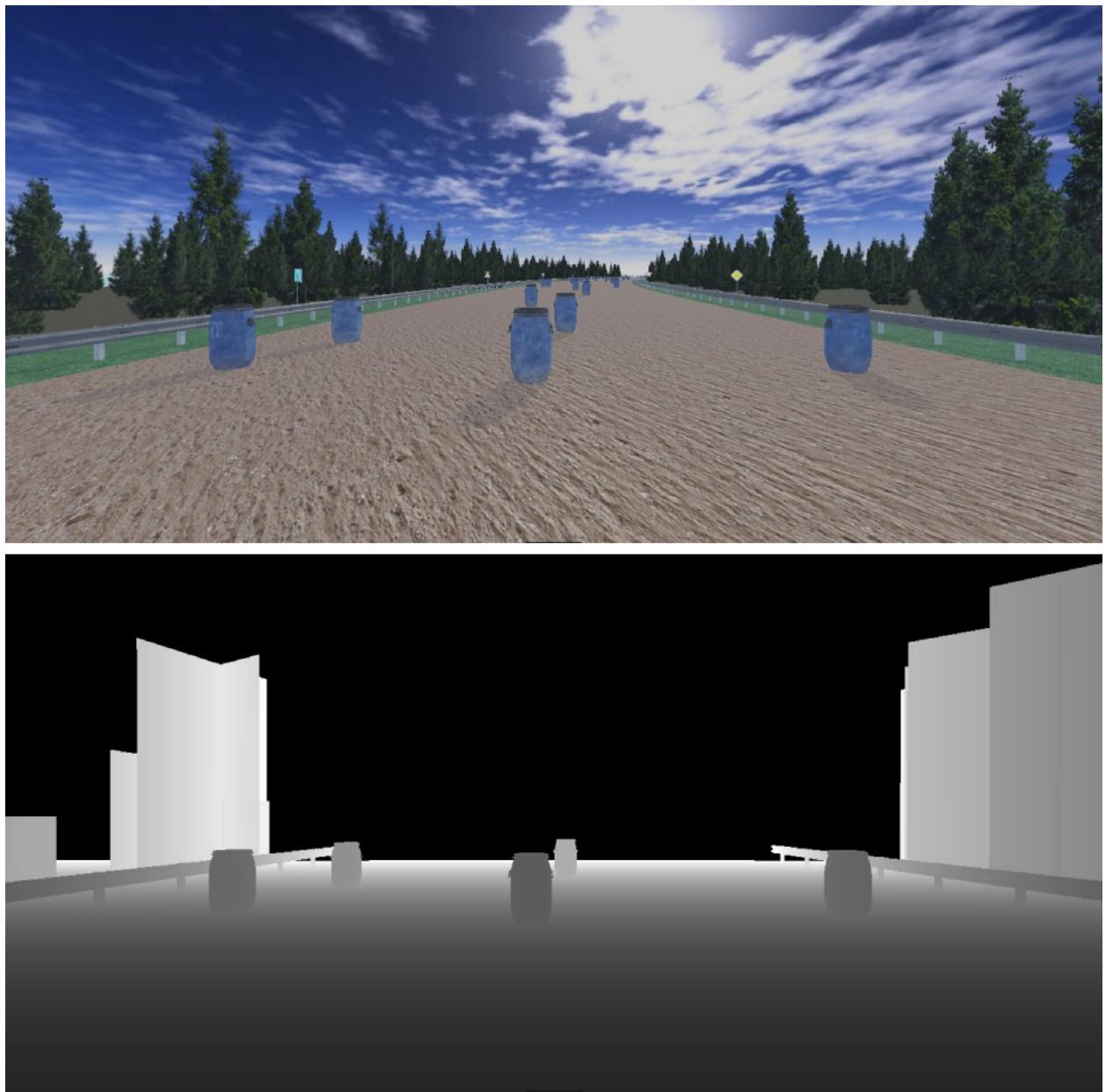


Рис. 7. Карта глубины (снизу) с передней стереокамеры в Webots

5. **image** – итоговое единое изображение кругового обзора размерностью `bev_parameters.total_height` на `bev_parameters.total_width` (см. Лист. 1) и формата `np.uint8`;
6. **@front_left** – одно из множества свойств `@property` класса `BirdsEyeView`, которое возвращает переднюю левую часть `image`, срезая его по `[:yt, :xl]`;
7. **@right_central** – одно из множества свойств `@property` класса `BirdsEyeView`, которое возвращает правую центральную часть `image`, срезая его по `[yt:yb, xr:]`.

Весь набор этих свойств: `@front_central`, `@front_central_blind`, `@front_right`, `@back_left`, `@back_central`, `@back_right`, `@left_central` и `@central` используется для копирования в эти части единого изображения `image` других, в том числе и уже объединённых методом `merge` изображений из `frames`. Делается это, в свою очередь, в методе `stitch`;

8. `init(images, load_weights_and_masks)` – сохраняет `images`, которые являются тем самым словарём `images_projected`, передаваемым в метод `get_weight_matrix` в узле `projection_weight_matrices_node` (см. Рис. 5 выше), во `frames`.

Инициализирует `weights` и `masks` как `None`, однако, если `load_weights_and_masks` передаётся как `True` (по умолчанию `False`), вызывается одноимённый метод ниже по списку.

Также, через метод `np.zeros`, создаёт `image` заданной размерности и формата (см. пункт 5 выше);

9. `load_weights_and_masks()` – загружает через метод `np.load` уже ранее посчитанные и сохранённые в файлы формата `npz` весовые матрицы `G` и маски `M`, после чего записывает их в `weights` и `masks` соответственно. Это позволяет избежать их пересчёта каждый раз, когда требуется получить BEV из новых изображений;
10. `get_weights_and_masks()` – ключевой метод рассматриваемого компонента, вызов которого запускает цепочку шагов 3-5 (см. список ниже из 6-ти шагов после этого перечисления) из алгоритма формирования единого изображения из пяти отдельных, разбитых при этом на ещё более мелкие части набором методов `get_upper_part`, `b_get_central_part` и т. п. (см. пункты 16-17 ниже);

11. **`luminance_balance()`** – устраняет различия в яркости на итоговом изображении кругового обзора, также разбитого на части, по следующей базовой идеи: каждая из пяти виртуальных видеокамер в Webots выдаёт трёхканальное изображение в формате BGR. По соотношению яркостей пяти таких изображений в их перекрывающихся областях рассчитывается 15 коэффициентов по одному на канал, которые затем умножаются на каждый из этих 15-ти каналов и объединяются для формирования скорректированного изображения. Таким образом, слишком яркие каналы будут затемнены, чтобы их коэффициенты получились равными меньше 1, а слишком тёмные – осветлены, с коэффициентами больше 1;
12. **`merge(image_1, image_2, i)`** – по формуле (11) ниже сшивает перекрывающиеся области `image_1` и `image_2` изображений из `frames`, используя для этого i -ый элемент (весовую матрицу) из списка `weights`;
13. **`stitch()`** – с использованием набора методов `get_upper_part`, `b_get_central_part` и т. п., дробит смежные изображения из `frames` на несколько частей – их общих перекрывающихся областей – после чего передаёт их в качестве аргументов в метод `merge` и копирует полученный результат в одну из частей `image` по множеству свойств `@property` (см. пункты 6-7 выше) в зависимости от расположения исходных частей (например, левая часть переднего изображения, объединённая с верхней частью левого изображения, копируется в `@front_left` из того же пункта 6);
14. **`white_balance()`** – устраняет различия в цветовом балансе на итоговом изображении кругового обзора по идеи, схожей с балансировкой яркости в `luminance_balance` (см. пункт 11): `image` разбивается на три отдельных канала: B, G и R, по каждому из которых затем считается его среднее значение через метод `pr.mean`.

Получившиеся коэффициенты суммируются и масштабируются, а три отдельных канала снова объединяются в один через метод cv2.merge;

15. **add_ego_vehicle_and_track_obstacles()** – накладывает поверх итогового изображения кругового обзора область «слепой» зоны (см. Рис. 9 ниже), которую не покрывает область видимости ни одной из камер, а также следующую иконку эго-автомобиля прямо по центру:



Рис. 8. Иконка эго-автомобиля

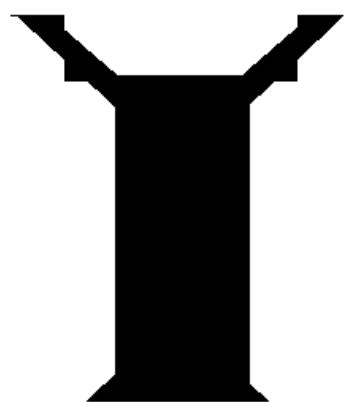


Рис. 9. Область «слепой» зоны

Кроме этого, данный метод является частью сегментированной локальной карты (см. следующий раздел «**Сегментированная локальная карта**») и используется в ней для визуализации obstacle_corners через cv2.polyline, obstacle_centers через cv2.line и obstacle_distances_m через cv2.putText на image (см. пункт 5) для каждого изображения из frames (см. пункт 2);

16. **get_upper_part(image)** – возвращает верхнюю часть какого-либо изображения image из frames (не путать наименование аргумента данного метода с переменной класса image из пункта 5!), срезая его по [:yt, :];
17. **b_get_central_part(image)** – возвращает центральную часть исключительно заднего (b – back) изображения image из frames, срезая его по [:, xl:xr]. Оставшийся набор аналогичных методов, принцип работы которых заложен в их название, следующий: **get_lower_part, lr_get_central_part, get_left_part, get_right_part, f_get_central_part_blind** и **f_get_central_part**.

Формирование же самого единого изображения из пяти отдельных осуществляется по следующему алгоритму:

1. Видеокамеры попарно располагаются таким образом, чтобы каждые две смежные из них хотя бы частично захватывали одну из угловых областей, помеченных на Рисунке 26. Так, например, для левой и передней камеры это будет левая верхняя область, а для задней и правой – нижняя правая. Обводка вокруг 3D-модели автомобиля на всём том же Рисунке 26 – это «слепая» зона, которая не покрывается обзором камер;
2. Для каждого из устройств, по четырём ключевым точкам, выбираемым вручную, рассчитывается матрица проекции, которая преобразует исходное перспективное изображение (верхнее окно на Рисунке 10) в него же, но с видом сверху:

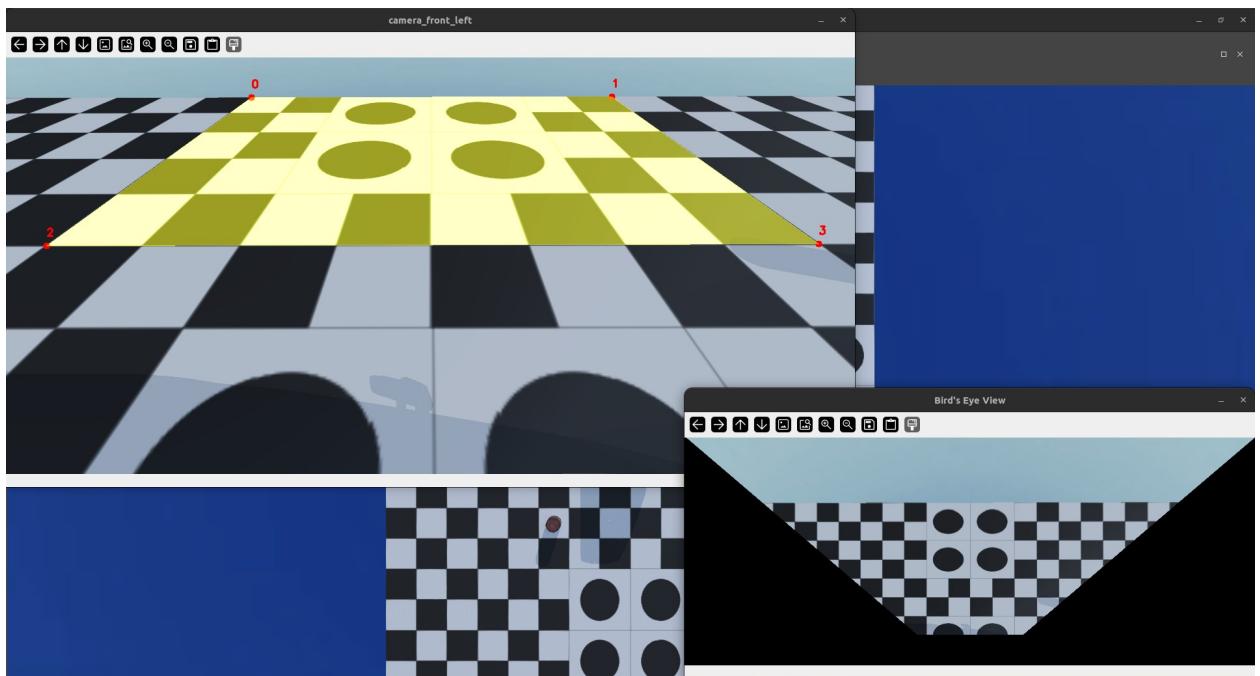


Рис. 10. Результат применения проекционной матрицы (нижнее окно)

3. Затем для каждой пары смежных видеокамер на этих изображениях находится та самая перекрывающаяся область через вызов методов cv2.bitwise_and и cv2.bitwise_not, после чего из неё создаётся бинарная маска через вызов методов cv2.threshold и cv2.dilate.

На Рисунке 11 ниже и далее до конца данного раздела в качестве примеров будут представлены сторонние изображения из документации оригинальной работы [20], на которой была основана описываемая уже здесь система двухмерного кругового обзора:

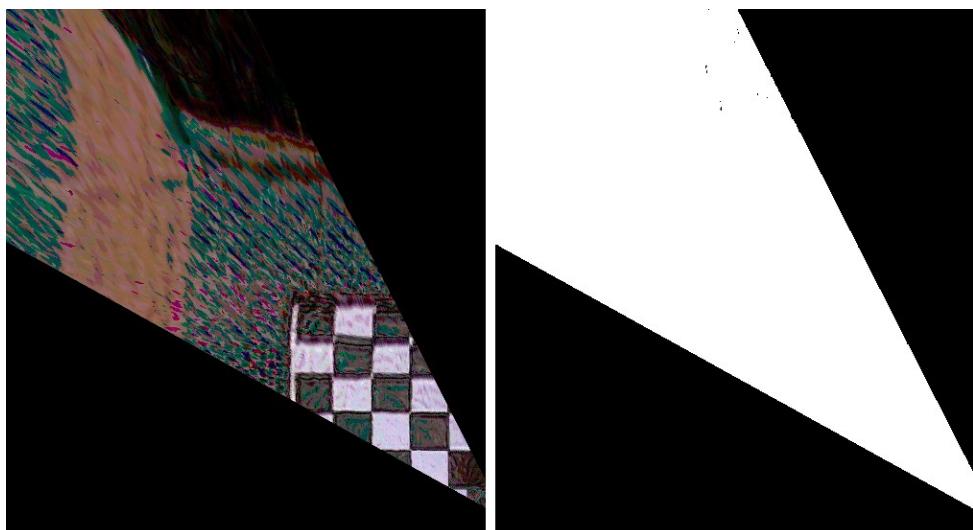


Рис. 11. Пример перекрывающейся области (слева) и её бинарной маски (справа)

4. Через применение методов cv2.findContours и cv2.approxPolyDP на всё тех же изображениях находятся контуры за пределами перекрывающейся области и относятся к той или иной камере.

Для каждого пикселя в общей части двух изображений рассчитывается расстояние до двух найденных ранее полигонов через вызов метода cv2.pointPolygonTest и его вес w по следующей формуле:

$$w = d_A^2 / (d_A^2 + d_B^2) \quad (10)$$

где: d_A, d_B – расстояния до полигона А (например, передняя камера) и полигона В (например, левая камера). Если пиксель составляет переднее изображение, то расстояние от него до полигона В будет больше, увеличивая таким образом общее значение веса. Пиксели за пределами общей части будут иметь веса равные 0 для левой камеры и 1 для передней;

5. Используя описанную выше информацию о весах каждого пикселя в перекрывающейся и неперекрывающейся областях изображений с любых двух смежных камер, составляется весовая матрица G , значения которой лежат в диапазоне от 0 до 1 и которая представлена на следующем Рисунке 12 вместе с контурами (полYGONами), упомянутыми ранее:

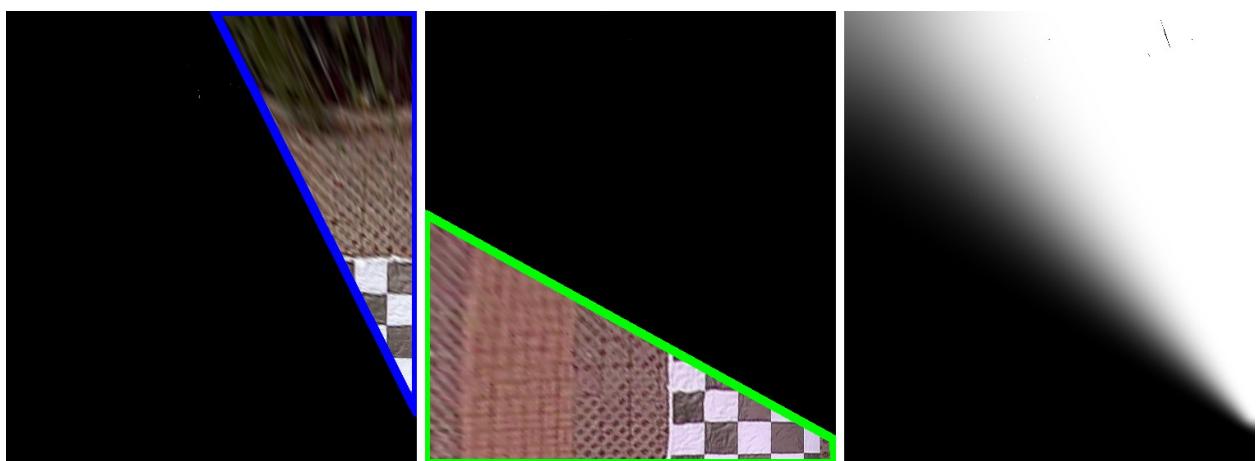


Рис. 12. Полигон А, полигон В и весовая матрица G

Объединённое итоговое изображение *fusedImage*, с видом сверху, для левой и передней видеокамер может быть получено как:

$$fusedImage = frontImage * G + (1 - G) * leftImage \quad (11)$$

где: *frontImage*, *leftImage* – исходные изображения с соответствующих камер после применения к ним матриц проекции (см. Рис. 10).

По аналогии шаги 3-6 проделываются для переднего и правого, правого и заднего, заднего и левого устройств, что на выходе даёт четыре весовые матрицы *G0-G3*, которые вместе с четырьмя масками перекрывающихся областей *M0-M3* объединяются в два единых изображения формата RGBA (см. Рис. 13) и сохраняются в файлы *weights.npy* и *masks.npy* (см. Рис. 5) соответственно для дальнейшей репродукции итогового результата:

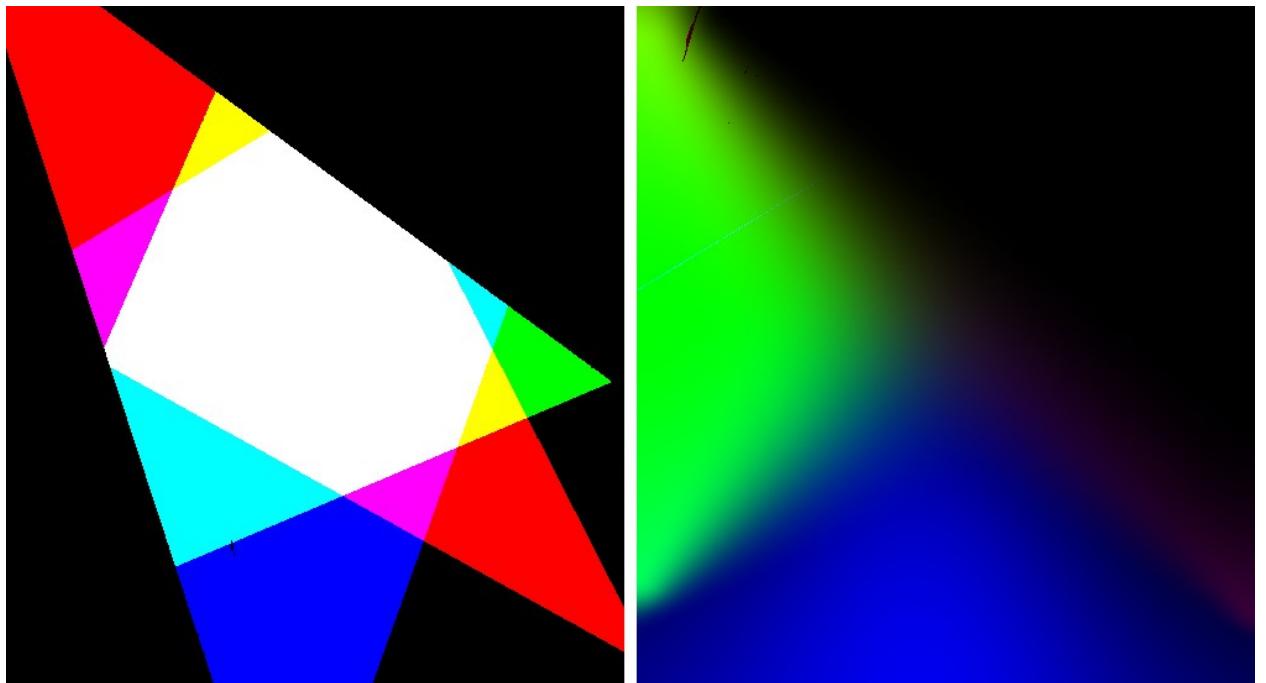


Рис. 13. Объединённые маски *M* и веса *G*

6. В заключение, и при необходимости, на итоговом изображении двухмерного кругового обзора (см. Рис. 32) производится устранение различий в яркости и цветовом балансе.

2.4 Сегментированная локальная карта

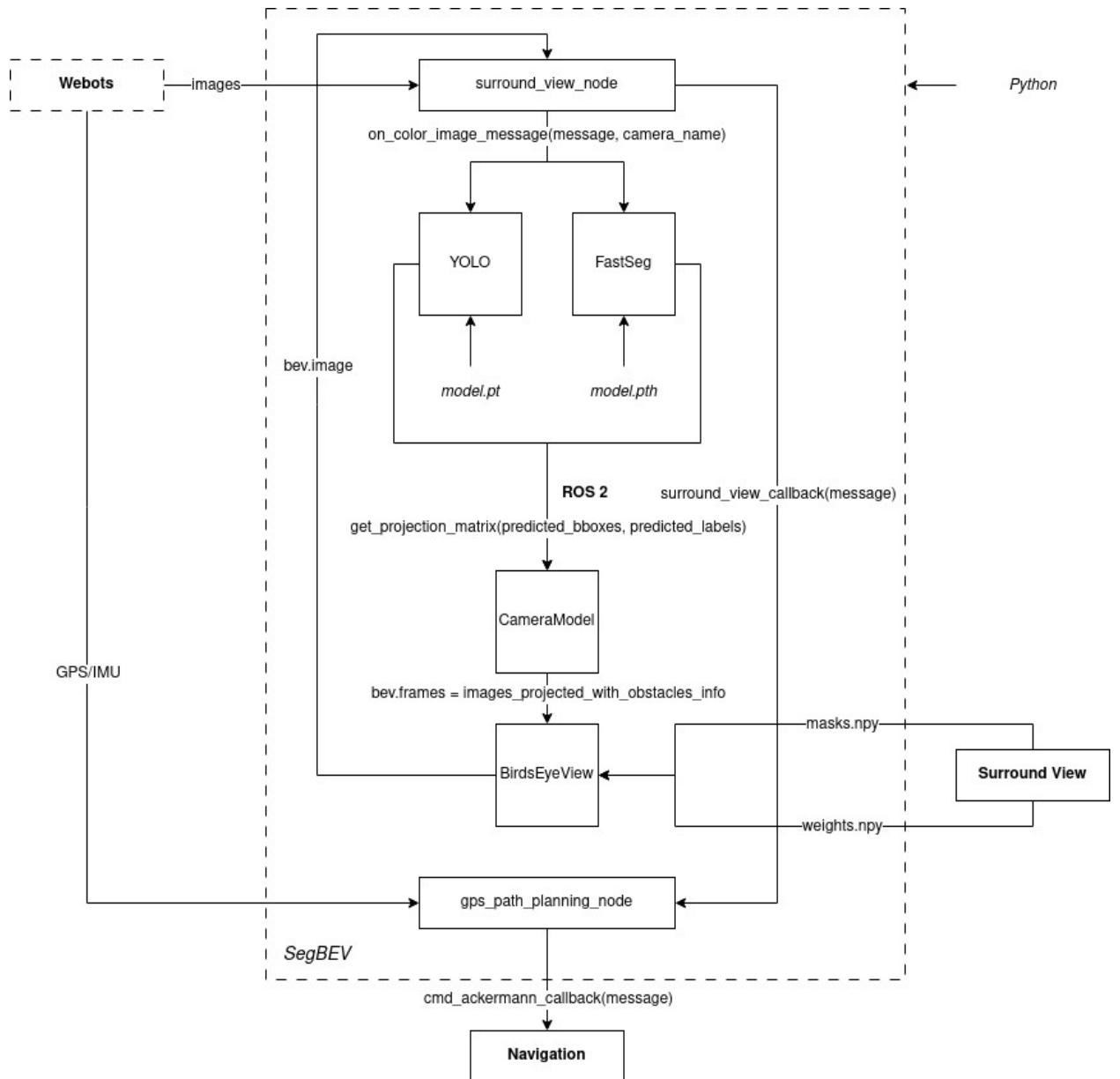


Рис. 14. Архитектура модуля «SegBEV»

Из Webots с каждой стереокамеры изображения `images` синхронно поступают в узел под названием «`surround_view_node`» с использованием таких встроенных в ROS 2 классов как: `Subscriber`, `TimeSynchronizer` (`ApproximateTimeSynchronizer`) и их методов `registerCallback`.

Далее, если в строке camera_name аргумента метода on_color_image_message содержится приписка 'depth' – это значит, что поступившее изображение является глубинным и его необходимо преобразовать в формат 32-х-битного числа с плавающей запятой и одним каналом (32FC1) через метод CvBridge().imgmsg_to_cv2 для дальнейшего извлечения из него информации о расстоянии до препятствий.

В случае же, если такая приписка в camera_name отсутствует, то байтовое изображение message, через всё тот же метод класса CvBridge, преобразуется в формат rgb8 и передаётся в таком виде на вход двум «нейронкам» – YOLO и FastSeg, которые предварительно были обучены, а их веса загружены из файлов model.pt и model.pth соответственно. Немного более подробно про сам процесс их обучения расписано в **Эксперименте №1** раздела 3.3 «**Сегментированная локальная карта**» главы 3.

Модели делают свои предсказания на новом цветном изображении, после чего первая из них выдаёт predicted_bboxes (предсказанные YOLO ограничивающие рамки препятствий), а вторая – predicted_labels (предсказанные FastSeg классы для каждого пикселя на изображении), которые затем передаются в метод get_projection_matrix (см. пункт 18 раздела 2.2 выше). Выходной результат этого метода – словарь images_projected_with_obstacles_info – затем присваивается переменной frames компонента «BirdsEyeView» (см. пункт 2 раздела 2.3 выше), после чего, с использованием всей этой информации, в модуле «Surround View» начинается процесс формирования единого изображения кругового обзора из пяти отдельных, на которых нейронные сети сделали свои предсказания.

В данном случае, поскольку маски перекрывающихся областей (masks.npy) и весовые матрицы (weights.npy) для BEV уже были посчитаны ранее, всё в том же разделе 2.3 «**Система двухмерного кругового обзора**», то их можно переиспользовать и на новых данных для ускорения вычислений.

Перейдём теперь к подробному рассмотрению узла автономной спутниковой навигации под названием «gps_path_planning_node», который принимает в себя `bev.image` в виде message через срабатывание метода обратного вызова `surround_view_callback`, а также данные с таких сенсоров как GPS и IMU из Webots (см. Рис. 14 выше).

В первом случае, изображение кругового обзора требуется данному узлу для его визуализации (см. Рис. 44 ниже) и отладки, поскольку `surround_view_callback` вызывает в себе другой метод под названием «`draw_path_on_surround_view`», который наносит на SegBEV заданный списком виртуальных GPS-координат маршрут движения БНТС в виде точек пути через метод `cv2.circle`, а также требуемый и фактический вектора направления движения ТС через метод `cv2.arrowedLine`. Информация о дистанции до текущей точки маршрута и о расстоянии до каждого из окружающих транспортное средство препятствий наносится на SegBEV через уже знакомый метод `cv2.putText`.

Во втором случае, данные для визуализации и отладки рассчитываются в самом `gps_path_planning_node` по довольно простой математике. Так, по компасу (изначальная альтернатива IMU), из топика «/compass» извлекаются координаты виртуального магнитного поля Земли по осям X, Y и Z, после чего курс `yaw` этого-автомобиля в радианах относительно севера рассчитывается по ним через следующую формулу:

$$yaw = \arctan(y, x) \quad (12)$$

, а его вектор направления `egoVehicleVector` через:

$$egoVehicleVector = [\cos(yaw), \sin(yaw)] \quad (13)$$

Позиция автомобиля на сцене №3 в Webots (см. Рис. 34 ниже) определяется просто по его метровым GPS-координатам из топика «/gps». Так же собираются и точки маршрута его движения, путём перемещения робота по сцене с использованием ручного управления с клавиатуры и последующего добавления этих координат в список `route` (см. Лист. 2 ниже):

```
# active, latitude (Y), longitude (X), distance
self.route = [
    ...
    [False, -28.47221484907397, -1.2501082404662278, 0.0],
    ...
]
```

Листинг 2. Список виртуальных GPS-координат маршрута движения БНТС

Каждые 0.2 секунды в узле `gps_path_planning_node` срабатывает так называемый ROS-таймер, который вызывает метод `navigate`, заставляющий Mercedes-Benz Sprinter ехать, учитывая построенный путь, по следующей логике:

```
self.route[self.current_route_point_index][0] = True # Выбираем текущую точку маршрута движения

current_latitude, current_longitude = self.ego_vehicle_position # Где мы
_, target_latitude, target_longitude, _ = self.route[self.current_route_point_index] # Куда нам надо

dx = target_longitude - current_longitude
dy = target_latitude - current_latitude

# Рассчитываем дистанцию до выбранной точки пути
distance_to_target_route_point = math.sqrt(dx**2 + dy**2)
self.route[self.current_route_point_index][3] = distance_to_target_route_point

# Если она менее, чем в 5-ти метрах от задней оси эго-автомобиля
if distance_to_target_route_point < 5.0:
    self.route[self.current_route_point_index][0] = False
    self.route[self.current_route_point_index][3] = 0.0
    self.current_route_point_index += 1 # Начинаем движение к следующей точки из списка (см. Лист. 2)
return

# Иначе, рассчитываем угол и скорость (ниже) до выбранной точки пути
angle_to_target_route_point = get_vectors_angle(self.ego_vehicle_vector, [dy, dx])

# с учётом скоростных и кинематических ограничений БНТС (см. Лист. 11 в Приложении А ниже)
self.drive_command.speed = min(self.max_speed, distance_to_target_route_point * 2.5) # Pk = 2.5
self.drive_command.steering_angle = max(
    -self.max_steering_angle,
    min(self.max_steering_angle, math.degrees(angle_to_target_route_point)))
)

# Отправляем управляющую команду на контроллер движения в модуль «Navigation» (см. Рис. 14 выше)
self.cmd_ackermann_publisher.publish(self.drive_command)
```

Листинг 3. Основная часть логики автономной спутниковой навигации

Угол до выбранной точки пути angle_to_target_route_point между двумя 2D-векторами ego_vehicle_vector (\vec{a}) и [dy, dx] (\vec{b}) рассчитывается по следующим формулам:

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y = |\vec{a}| |\vec{b}| \cos \theta \quad (14)$$

где: $\cos \theta$ определяет величину косинуса угла между векторами,

$$\det(\vec{a} \cdot \vec{b}) = a_x b_y - a_y b_x = |\vec{a}| |\vec{b}| \sin \theta \quad (15)$$

где: $\sin \theta$ определяет величину синуса угла между векторами, а

$\det(\vec{a} \cdot \vec{b})$ – его знак (ориентацию) по следующей логике:

$$\det(\vec{a} \cdot \vec{b}) > 0 - \vec{b} CCW \vec{a} \text{ и } \det(\vec{a} \cdot \vec{b}) < 0 - \vec{b} CW \vec{a},$$

$$\theta = \arctan(\det(\vec{a} \cdot \vec{b}), \vec{a} \cdot \vec{b}) \quad (16)$$

где: θ – искомый угол в диапазоне $[-\pi, \pi]$.

2.5 Глобальная карта и планирование маршрута движения беспилотного наземного транспортного средства

2.5.1 Глобальная карта

Одометрия – это оценка перемещения робота в физическом пространстве с течением времени за счёт использования информации о движении его приводов и прочих данных, поступающих с пригодных для этого сенсоров (например, изображения с камер для реализации визуальной одометрии).

Как правило, основываясь именно на различных типах одометрии и работают большинство современных алгоритмов одновременной локализации и картирования (англ. simultaneous localization and mapping) или попросту – SLAM.

Локализация – это оценка месторасположения (положения и вращения) робота на карте (только ещё формируемой или уже построенной) в конкретный момент времени за счёт использования, например, данных с лидара и последующего рассчёта одометрии по ним.

Картирование – это процесс непосредственного построения карты окружающего пространства вокруг робота по мере его перемещения по ней и, как следует из двух определений выше, по мере обновления одометрии, а вместе с ней и локализации робота на карте.

Существует множество различных подходов к реализации SLAM, некоторые из которых были упомянуты в подразделе 1.5.1 «**Анализ существующих подходов**» главы 1, однако, как это было определено там же, в подразделе 1.5.2 «**Описание используемых технологий и инструментов**» использоваться в данном решении будут следующие из них:

1. **Одометрия**: очень точные GPS-данные для определения положения БНТС совместно с не менее идеальными данными от блока IMU для определения вращения (курса);
2. **Локализация**: LaserScan от твёрдотельного (статического) лидара совместно с её последующим уточнением за счёт данных одометрии;
3. **Картирование**: тот же LaserScan от лидара, что и при локализации.

Теперь, после пояснения этих трёх основных терминов, составляющих большинство современных алгоритмов построения глобальных карт, можно приступить к подробному описанию архитектуры модуля «Navigation», архитектура которого представлена на следующей диаграмме:

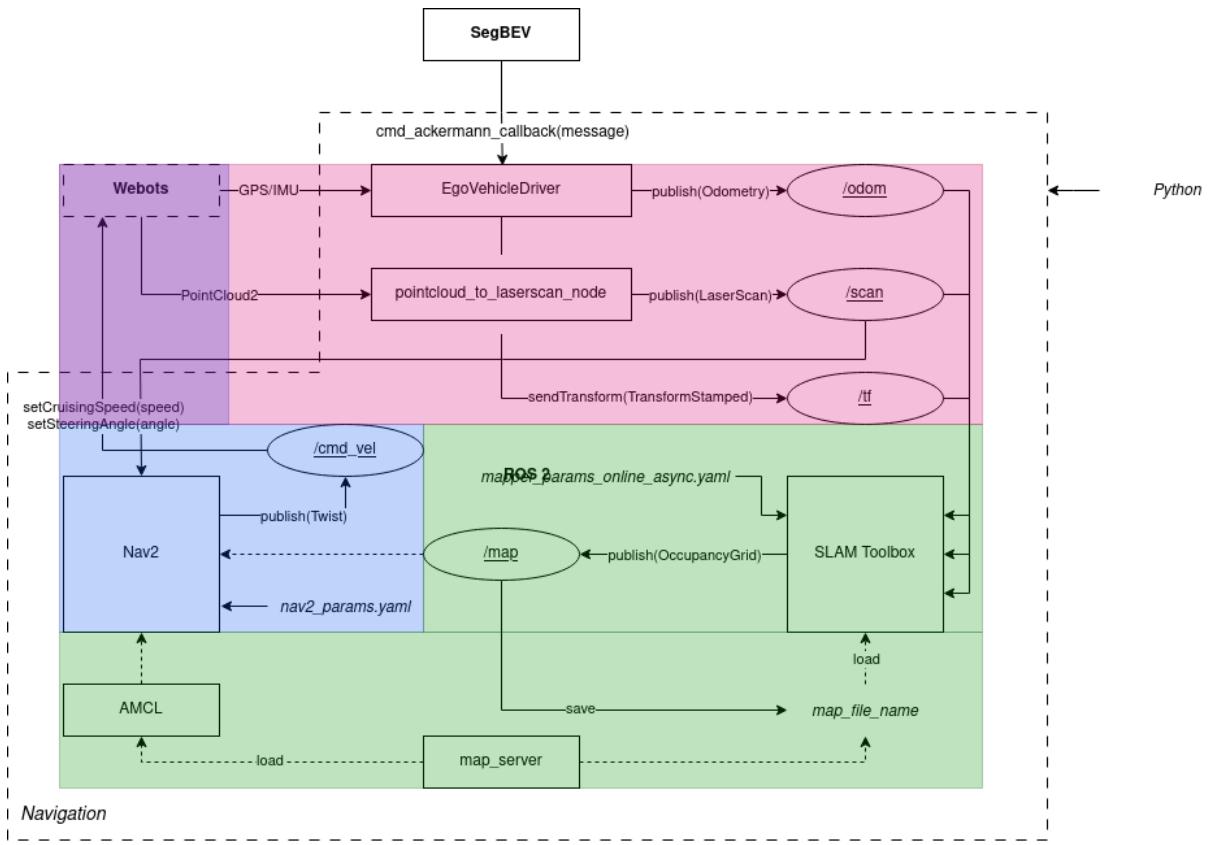


Рис. 15. Архитектура модуля «Navigation»

Из модуля «SegBEV» в модуль «Navigation» поступает информация с сегментированной локальной карты в виде команды управления, полученной от узла `gps_path_planning_node`, логика работы которого была рассмотрена ранее (см. Рис. 14).

Переходя внутрь самого модуля, в глаза сразу же бросаются цветовые разграничения (см. Рис. 15), где блок фиолетово-розового цвета отвечает за одометрию и потоки данных, зелёный – за локализацию и картирование, а синий – за навигацию и управление. Визуализирует все эти процессы уже знакомый робототехнический симулятор Webots, который отправляет в ROS 2 данные с GPS, IMU и лидара (PointCloud2), а принимает от фреймворка управляющие команды в виде скорости автомобиля в километрах в час и угла поворота его колёс в градусах за счёт вызова методов `setCruisingSpeed` и `setSteeringAngle` в компоненте контроллера робота под названием «EgoVehicleDriver», содержимое которого представлено на следующей диаграмме:

<i>robot</i>	<i>init(webots_node)</i>
<i>drive_command</i>	<i>point_cloud_callback(message)</i>
<i>compass</i>	<i>compass_publisher_callback()</i>
<i>compass_message</i>	<i>odom_callback(message)</i>
<i>imu</i>	<i>imu_odom_publisher_callback()</i>
<i>quaternion_message</i>	<i>gps_publisher_callback()</i>
<i>gps</i>	<i>cmd_ackermann_callback(message)</i>
<i>gps_message</i>	<i>stop()</i>
<i>lidar</i>	<i>cmd_vel_callback(message)</i>
<i>tfs</i>	<i>step()</i>
<i>odom</i>	<i>EgoVehicleDriver</i>

Рис. 16. Содержимое компонента «EgoVehicleDriver»

1. **robot** – объект робота в Webots [21], с помощью которого можно программно управлять им через код, например, путём вызова метода `setDippedBeams`, который включает или выключает (в зависимости от их текущего состояния) ближний свет и противотуманные фары на это-автомобиле, а также методов `setCruisingSpeed`, `setSteeringAngle`, уже упомянутых ранее, и `getDevice`, который позволяет извлечь объект сенсора для последующего получения данных с него;
2. **drive_command** – объект ROS-сообщения (msg) [22] класса `AckermannDrive` [23], который предназначен для хранения команд управления и имеет следующую структуру:
 - `float32 steering_angle`: угол поворота колёс (радианы) – преобразуется в градусы для более удобной интерпретации и используется в решении;
 - `float32 steering_angle_velocity`: скорость поворота колёс (радианы/с) – не используется в решении;

- float32 speed: целевая скорость движения (м/с) – преобразуется в км/ч для более удобной интерпретации и используется в решении;
 - float32 acceleration: целевое ускорение движения ($\text{м}/\text{с}^2$) – не используется в решении;
 - float32 jerk: отклик ускорения ($\text{м}/\text{с}^3$) – не используется в решении. Нулевое значение будет пытаться изменить ускорение при первой возможности, положительное значение – на желаемую абсолютную величину в любом направлении (увеличение или уменьшение).
3. **compass** – объект компаса (виртуальная частота работы – 5 Гц), который используется как альтернативный способ определения ориентации (курса) БНТС относительно условного магнитного севера Земли в мире Webots.
 4. **compass_message** – объект ROS-сообщения класса MagneticField [24], который предназначен для хранения относительных координат магнитного поля и имеет следующую структуру:
 - uint32 seq: идентификатор текущего сообщения – является частью заголовка [25] ROS-сообщения;
 - time stamp: время получения текущего сообщения, прошедшее с начала «эпохи» (секунды и наносекунды с 1-го января 1970 года) – также является частью заголовка;
 - string frame_id: элемент (*фрейм* или имя координатной системы) робота, к которому будут привязываться по положению, ориентации и частоте поступающие с сенсора данные – часть заголовка, который имеет одинаковую структуру для абсолютно любого типа сообщений в ROS 2, в связи с чем дальше по тексту он будет указываться просто как std_msgs/Header header;

- geometry_msgs/Vector3 magnetic_field: float64-вектор, содержащий в себе компоненты магнитного поля в Теслах по осям x, y и z в системе координат, заданной frame_id;
 - float64[9] magnetic_field_covariance: *ковариационная матрица* магнитного поля – представляет собой неопределённость измерений (шум, дрейф и т. п.). В решении напрямую не используется, а это значит, что принимает значения -1 (*ковариация неизвестна*) по умолчанию.
5. **imu** – объект инерциального измерительного блока (англ. Inertial Measurement Unit – IMU). Виртуальная частота работы – 31.25 Гц), который используется как основной способ определения ориентации (курса) БНТС по набору данных со встроенных в блок датчиков: акселлерометра, гироскопа и магнитометра;
 6. **quaternion_message** – объект ROS-сообщения класса Quaternion [26], который предназначен для хранения ориентации в формате кватерниона и имеет следующую структуру:
 - float64 x, y, z, w: координаты кватерниона, который представляет из себя математический объект, хранящий в себе информацию о вращении и ориентации в трёхмерном пространстве другого объекта вокруг соответствующих осей X, Y и Z на заданное значение угла W.
 7. **gps** – объект системы глобального позиционирования (англ. Global Positioning System – GPS). Виртуальная частота работы – 3.9 Гц), который используется для определения глобальных метровых GPS-координат эго-автомобиля на сцене №3 в Webots (см. Рис. 34);
 8. **gps_message** – объект ROS-сообщения класса NavSatFix [27], который предназначен для хранения географических глобальных координат и имеет следующую структуру:

- std_msgs/Header header: заголовок сообщения;
 - sensor_msgs/NavSatStatus status: один из статусов и режимов работы виртуальной системы GPS – в решении напрямую не используется;
 - float64 latitude: широта (градусы) – сенсор в Webots выдаёт значения уже в метрах, как они дальше и обрабатываются;
 - float64 longitude: долгота (градусы) – аналогично широте;
 - float64 altitude: высота (м) – не используется в решении, поскольку автомобиль всегда движется исключительно по плоской земле без всяких перепадов высот;
 - float64[9] position_covariance: по аналогии с ковариационной матрицей магнитного поля, ковариационная матрица текущей позиции – в решении напрямую не используется;
 - uint8 position_covariance_type: один из типов ковариации выше – в решении напрямую не используется;
9. **lidar** – объект лидара (виртуальная частота работы – 10 Гц). В симуляторе сам сенсор расположен на капоте под лобовым стеклом Mercedes-Benz Sprinter (см. Рис. 17 ниже) и обладает следующими характеристиками, эмулируя собой таким образом реальную модель лидара LS Lidar CB64S1 [28]:
- Количество горизонтальных каналов сканирования: 500 – влияет на горизонтальную плотность данных;
 - Горизонтальный угол обзора: 180 (градусы);
 - Вертикальный угол обзора: 40 (градусы) – угол между первым и последним слоем;
 - Количество слоёв (вертикальных лучей): 64 – влияет на вертикальную плотность данных;
 - Минимальная дальность измерения: 0.1 (м);
 - Максимальная дальность измерения: 100 (м).



Рис. 17. Лидар на эго-автомобиле в Webots и его характеристики

10. **tfs** – объект ROS-сообщения класса `TransformStamped` [29], который предназначен для хранения трансформации (перемещения и вращения) между двумя фреймами и имеет следующую структуру:
 - `std_msgs/Header header`: заголовок сообщения;
 - `string child_frame_id`: название дочернего фрейма робота, который будет привязываться к `string frame_id` из заголовка;
 - `geometry_msgs/Vector3 translation`: перемещение `child_frame_id` относительно `frame_id` по `float64`-осям x, у и z;
 - `geometry_msgs/Quaternion rotation`: вращение `child_frame_id` относительно `frame_id` по `float64`-значениями x, y, z и w в формате кватерниона – совместно с предыдущим полем перемещения входит в `geometry_msgs/Transform`.
11. **odom** – объект ROS-сообщения класса `Odometry` [30], который предназначен для хранения оценки положения, а также скорости (одометрии) робота и имеет следующую структуру:
 - `std_msgs/Header header`: заголовок сообщения;
 - `string child_frame_id`: название дочернего фрейма робота, который будет привязываться к `string frame_id` из заголовка;

- geometry_msgs/Point position: позиция child_frame_id относительно frame_id, задаваемая float64-осяями x, у и z – рассчитанное примерное положение транспортного средства в системе координат frame_id;
 - geometry_msgs/Quaternion orientation: ориентация child_frame_id относительно frame_id, задаваемая float64-значениями x, y, z и w в формате кватерниона – рассчитанный примерный курс транспортного средства в системе координат frame_id. Совместно с предыдущим полем позиции входит в geometry_msgs/Pose;
 - float64[36] covariance: ковариация позы – в решении напрямую не используется и совместно с geometry_msgs/Pose входит в geometry_msgs/PoseWithCovariance;
 - geometry_msgs/Vector3 linear: линейная скорость child_frame_id относительно frame_id, задаваемая float64-осяями x, у и z – рассчитанная примерная скорость движения ТС в системе координат frame_id;
 - geometry_msgs/Vector3 angular: угловая скорость child_frame_id относительно frame_id, задаваемая float64-осями x, у и z – рассчитанная примерная скорость вращения ТС в системе координат frame_id. Совместно с предыдущим полем скорости входит в geometry_msgs/Twist;
 - float64[36] covariance: ковариация скоростей движения – в решении напрямую не используется и совместно с geometry_msgs/Twist входит в geometry_msgs/TwistWithCovariance;
12. **init(webots_node)** – инициализирует («включает») работу из встроенного в симулятор узла webots_node, а также запускает все, описанные выше сенсоры на нём;

13. **point_cloud_callback(message)** – вызывается каждый раз, когда лидар отправляет свои данные message, в формате PointCloud2, из Webots (топик «/ego_vehicle/lidar/point_cloud») в ROS 2, после чего переопубликовывает их в том же формате в топик «/cloud_in» как того требует пакет pointcloud_to_laserscan и его компонент pointcloud_to_laserscan_node, работа которого описана после завершения данного списка;
14. **compass_publisher_callback()** – вызывается через каждые 0.2 секунды, чтобы получить данные с компаса, путём вызова метода getValues у его объекта, и переопубликовать заполненный ими compass_message (см. пункт 4 выше), в формате MagneticField, в топик «/compass» для дальнейшей обработки этих данных алгоритмами;
15. **odom_callback(message)** – вызывается один раз, когда метод ниже отправляет свои данные message, в формате Odometry, в топик «/odom», после чего переопубликовывает их в формате PoseWithCovarianceStamped [31] в топик «/initialpose» для AMCL и уничтожается. Для чего это делается и как работает AMCL рассмотрено в конце текущего подраздела;
16. **imu_odom_publisher_callback()** – вызывается через каждые 0.032 секунды, чтобы получить данные с «инерциалки», путём вызова метода getQuaternion у её объекта, и переопубликовать заполненный ими quaternion_message (см. пункт 6), в формате Quaternion, в топик «/imu_quaternion» для дальнейшей обработки этих данных алгоритмами. Кроме этого, данные с IMU совмещаются с данными от GPS, и заполненные ими tfs (см. пункт 10), в формате TransformStamped, вместе с odom (см. пункт 11), в формате Odometry, отправляются в упомянутый выше топик «/odom» и «/tf». Таким образом, совмещение данных от инерциальной и глобальной спутниковой систем образуют одометрию БНТС;

17. **gps_publisher_callback()** – вызывается через каждые 0.256 секунды, чтобы получить данные с GPS, путём вызова метода `getValues` у его объекта, и переопубликовать заполненный ими `gps_message` (см. пункт 8), в формате `NavSatFix`, в топик «`/gps`» для дальнейшей обработки этих данных алгоритмами;
18. **cmd_ackermann_callback(message)** – вызывается каждый раз, когда метод `cmd_vel_callback`, или любой другой, отправляет свои данные `message`, в формате `AckermannDrive`, в топик «`/cmd_ackermann`», после чего вызывает метод ниже `stop`, в случае если пришедшие значения имеют некорректный формат. В ином случае, проверяет их на заданные через глобальные параметры (см. Лист. 11) скоростные и кинематические ограничения автомобиля и формирует `drive_command` (см. пункт 2);
19. **stop()** – вызывается если что-то пошло не так, заставляя этот автомобиль экстренно остановиться на месте;
20. **cmd_vel_callback(message)** – вызывается каждый раз, когда компонент «`Nav2`» отправляет управляющие команды, в формате `Twist` [32], в топик «`/cmd_vel`» (см. Рис. 15), после чего множит значение линейной скорости по оси X на коэффициент `speed_factor`, который может принимать произвольные значения в зависимости от участка, по которому в данный момент времени передвигается робот, а также преобразует значение угловой скорости по оси Z из радиан в градусы, множа его затем на одно из чисел (2 или 10), подобранные экспериментальным путём. Отправляет сформированное из этого сообщение `drive_command` в упомянутый выше топик «`/cmd_ackermann`»;

21. `step()` – вызывается единожды при создании объекта node узла `ego_vehicle_driver_node`, связанного с классом `EgoVehicleDriver` одноимённого компонента, после чего зацикливается за счёт вызова метода `rclpy.spin_once(node, timeout_sec)`, заставляя таким образом работать данный узел с задержкой в `timeout_sec` вплоть до завершения всей программы в целом.

Также, вызывает в себе методы управления `setCruisingSpeed(drive_command.speed)` и `setSteeringAngle(drive_command.steering_angle)`, однако, если управляемые команды не приходят в «`/cmd_ackermann`» более, чем секунду, `step` вызывает `stop` (см. пункт 19 выше) до тех пор, пока они не начнут поступать снова, или пока не будет завершена работа программы.

Компонент «`pointcloud_to_laserscan_node`» представляет из себя сторонний пакет, который используется в решении для преобразования формата лидарных данных из `PointCloud2` в формат `LaserScan` (см. Рис. 45), который получается на выходе работы `pointcloud_to_laserscan` и публикуется в топик «`/scan`» (см. Рис. 15). Делается это для корректной работы компонента «`SLAM Toolbox`», о котором далее и пойдёт речь.

После полного рассмотрения фиолетово-розового блока одометрии и потоков данных, а также синего блока навигации и управления, перейдём к разбору заключительного зелёного блока локализации и картирования, в котором можно выделить два ключевых компонента – это «`SLAM Toolbox`» и «`AMCL`» (см. Рис. 15), первый из которых отвечает за картирование, а второй – за локализацию.

В общих чертах, построение глобальной карты с использованием 2D SLAM библиотеки происходит следующим образом:

1. В конфигурационном файле под названием «`mapper_params_online_async.yaml`» прописывается соответствующий режим «`mapping`» (см. Лист. 4 ниже) и задаются прочие параметры работы инструмента, такие, как например названия фреймов одометрии, карты и робота, а также название упомянутого выше топика с `LaserScan`, чтобы SLAM Toolbox мог брать из него лидарные данные и строить по ним карту:

```
slam_toolbox:
ros__parameters:
solver_plugin: solver_plugins::CeresSolver
ceres_linear_solver: SPARSE_NORMAL_CHOLESKY
ceres_preconditioner: SCHUR_JACOBI
ceres_trust_strategy: LEVENBERG_MARQUARDT
ceres_dogleg_type: TRADITIONAL_DOGLEG
ceres_loss_function: None

odom_frame: odom # Данные для этого фрейма обеспечиваются топиками «/odom» и «/tf» (см. Рис. 15)
map_frame: map
base_frame: base_footprint
scan_topic: /scan
mode: mapping
```

Листинг 4. Минимальный набор параметров для корректной работы SLAM Toolbox

2. С учётом информации из этого конфига, запускается узел под названием «`async_slam_toolbox_node`» для так называемого асинхронного онлайн-картирования в режиме реального времени и по мере движения робота по строящейся карте. Оффлайн-режим, который предназначен для синхронной и более точной постобработки заранее записанных данных, можно запустить через узел `sync_slam_toolbox_node`, но в текущем решении этот функционал не используется;
3. Первое сообщение (скан) из топика «`/scan`» используется как начальная поза (ключевой кадр) и добавляется в граф SLAM как узел с координатами $(0, 0, 0)$;
4. Каждый новый скан сравнивается с предыдущим с помощью алгоритмов ICP ([англ.](#) Iterative Closest Point) [33] и Ceres Solver [34] для определения их смещения друг относительно друга.

В случае, если требуемая трансформация превышает порог по расстоянию ([англ.](#) translational threshold) или по углу ([англ.](#) rotational threshold), скан добавляется в общий граф как новый узел, а между ним и предыдущим узлом создаётся ребро с этой трансформацией;

5. Периодически происходит поиск замыканий цикла ([англ.](#) loop closure detection), который выполняет сравнение текущего кадра с историческим по временной или пространственной эвристике.

Если совпадение обнаружено – создаётся новое ребро между несоседними узлами и граф пересчитывается путём обновления поз для всех узлов в нём. Делается это для оптимизации и минимизации ошибок;

6. На основе всех позиций узлов и связанных с ними сканов строится карта занятости ([англ.](#) occupancy grid map), которая публикуется в формате OccupancyGrid [35] в топик «/map» (см. Рис. 15 выше) и может быть сохранена, для дальнейшего переиспользования, в набор файлов map_file_name нескольких форматов (.data, .pgm (см. Рис. 51 ниже), .posegraph и .yaml), каждый из которых содержит свой формат описания одной и той же только что построенной глобальной карты.

В общих чертах, дальнейшее применение глобальной карты и определение месторасположения робота на ней с использованием вероятностной системы локализации происходит следующим образом:

1. Запускается отдельный узел компонента «Nav2» под названием «map_server», который берёт описания из сохранённых ранее файлов (.yaml и .pgm) карты map_file_name, совмещает их и начинает публиковать Occupancy Grid, как до этого это делал SLAM Toolbox. Под капотом map_server реализует ROS-сервис под названием «load_map», который позволяет динамически загружать 2D-карту во время работы решения;

2. Запускается отдельный узел компонента «Nav2» под названием «amcl», который начинает локализовывать робота на только что загруженной map_server глобальной карте по следующему алгоритму:

1. Задаётся начальное распределение частиц равномерно по всей карте, либо ручным заданием позы путём её публикации в топик «/initialpose», как это было описано в пункте 15 из списка содержимого компонента «EgoVehicleDriver» выше.

Каждая такая трёхмерная частица – это следующая гипотеза:

$$p_i = (x, y, \theta) \quad (17)$$

2. Для каждой частицы делается предсказание на основе данных одометрии и согласно следующей модели её движения:

$$p_i^{(t)} = motionModel(p_i^{(t-1)}, u_t) \quad (18)$$

где: u_t – одометрия; $p_i^{(t-1)}$ – предыдущая позиция i -ой частицы;

3. Считается вес каждой частицы w_i для корректировки и отсеивания сделанных ранее предсказаний:

$$w_i = p(z_t \vee p_i^{(t)}) \quad (19)$$

где: z_t – данные в формате LaserScan с лидара в момент времени t ; $p_i^{(t)}$ – гипотетическое положение робота, заданное i -ой частицей в момент времени t ; $p(z_t \vee p_i^{(t)})$ – вероятность получить измерение z_t , если бы робот находился в позе $p_i^{(t)}$. То есть, насколько хорошо это положение объясняет полученные сенсорные данные;

4. Путём удаления части старых частиц и дублирования наиболее «увесистых» из них выполняется процесс перевыборки ([англ. resampling](#)) в случае, если эффективное количество частиц N_{eff} упало ниже заданного порога:

$$N_{eff} = \frac{1}{\sum w_i^2} \quad (20);$$

5. Финальное месторасположение робота (его положение и ориентация) определяется как среднее значение всех частиц или максимум по их плотности, после чего оно, в формате PoseWithCovarianceStamped, публикуется в топик «/amcl_pose».
3. Как правило, оба описанных выше узла запускаются одновременно со всем стеком фреймворка Nav2 и подключаются к так называемому Lifecycle Manager, как и другие узлы компонента «Nav2», что позволяет централизованно управлять их состоянием. Довольно же тонко настраивать каждый из этих узлов в отдельности позволяет конфигурационный файл nav2_params.yaml, содержимое которого представлено в Листинге 12 **Приложения А**.

2.5.2 Планирование маршрута движения БНТС

Сразу стоит отметить, что какие-либо алгоритмы планирования маршрута движения БНТС не были интегрированы и не обеспечивают его автономное движение по испытательному полигону в Webots с использованием сегментированной локальной карты, а стрелки и дистанция (крупные цифры) до текущей точки пути на карте (см. Рис. 44), это лишь визуализация заранее записанных виртуальных GPS-координат и движение по ним. Мелкие цифры возле центра каждой из пластиковых бочек – это визуализация информации о расстоянии от виртуальных стереокамер до каждого из препятствий, которая берётся из карт глубины (см. Рис. 7 выше).

Вместо этого, в решение была интегрирована готовая реализация одного глобального и двух локальных планировщиков из фреймворка для навигации Nav2, обеспечивающая автономное движение Mercedes-Benz Sprinter по испытательному полигону в Webots, но с использованием не сегментированной локальной карты, а карты глобальной, построенной с применением библиотеки SLAM Toolbox и описанной в предыдущем подразделе 2.5.1 «**Глобальная карта**».

Рассмотрим теоретический аспект работы этих планировщиков, а вместе с этим и компонента «Nav2» (см. Рис. 15) более подробно.

NavFn Planner

NavFn Planner (Navigation Function Planner, далее – *NFP*) [36] является классическим глобальным планировщиком на 2D-сетке формата OccupancyGrid (см. Рис. 46-47 ниже), в случае с Nav2.

Основа *NFP* по умолчанию – это реализация одного из вариантов алгоритма Dijkstra для поиска кратчайшего, но грубого пути от текущей позиции робота на глобальной карте до цели, причём без учёта его динамики и кинематики. Именно поэтому для этих целей он обычно используется вкупе с локальным планировщиком, который уже и обеспечивает движение по построенному *NFP* маршруту с учётом ограничений платформы. Описание двух из таких локальных планировщиков можно найти ниже по тексту.

С математической точки зрения, Navigation Function Planner с Dijkstra «под капотом» работает следующим образом:

1. Создаётся граф $G = (V, E)$, где: V (вершины графа) – все проходимые для робота ячейки глобальной карты, которые имеют стоимость 0 и, следовательно, не содержат в себе препятствия (ячейки со стоимостью, например, 100 и более); E – рёбра графа между соседними ячейками (4 прямых и 4 диагональных). Через такой параметр данного планировщика как `allow_unknown` (True по умолчанию), можно добавлять в V ячейки карты с неизвестной стоимостью, в которых по тем или иным причинам непонятно что находится – область, по которой можно проехать, или препятствие;
2. Вес w ребра между двумя соседними вершинами (клетками сетки) $u = (x_1, y_1)$ и $v = (x_2, y_2)$ задаётся как:

$$w(u, v) = d(u, v) * (1 + \alpha * c(v)) \quad (21)$$

где: $d(u, v)$ – евклидово расстояние между ячейками (1 – для прямых, $\sqrt{2}$ – для диагональных); 1 – базовый множитель чистого движения без препятствий; α – коэффициент, который определяет насколько сильно робот будет избегать потенциально опасных ячеек с высокой стоимостью. Регулируется через такой параметр фреймворка как `cost_scaling_factor` (10.0 по умолчанию, см. Лист. 12 в **Приложении А**); $c(v)$ – стоимость клетки из Costmap (см. левую часть на Рис. 53 ниже), в которую планируется переход;

3. Все узлы $v \in V$ графа G добавляются в приоритетную очередь Q в порядке возрастания значения $g(v)$, которое формируется для каждой из свободных ячеек глобальной карты по следующей логике:

$g(s)=0$ – накопленная стоимость перехода от старта s до него же (не путать со стоимостью ячейки и со стоимостью перехода из текущей ячейки в соседнюю!),

$g(v)=\infty$ – начальная накопленная стоимость перехода от старта s до всех остальных ячеек;

4. Пока Q не пуста, извлекаем из неё вершину u с минимальной $g(u)$, которая находится ближе всего к старту s ;
5. Рассматриваем все смежные (соседние) и проходимые вершины $v \in adj(u)$, в которые можно попасть из u за один шаг;
6. Считаем новую стоимость gn перехода из u в v :

$$gn=g(u)+w(u, v) \quad (22)$$

где: $g(u)$ – накопленная стоимость перехода от старта s до текущей ячейки u с учётом всех предыдущих шагов; $w(u, v)$ – стоимость перехода из текущей ячейки u в соседнюю к ней проходимую ячейку v С учётом стоимости второй из Costmap (не путать со стоимостью ячейки и с накопленной стоимостью всего пути!);

7. Если gn (новая стоимость всего маршрута от старта s до следующей в нём точки v) меньше $g(v)$ (ранее посчитанной стоимости всего маршрута от старта s до следующей в нём точки v), то:

1) в Q заменяем $g(v)$ на gn , поскольку теперь мы можем добраться до v за меньшую стоимость,

- 2) запоминаем, что предок v – это u , чтобы по достижению целевой точки всего глобального пути можно было добраться от неё до старта s и построить конечный вариант наиболее оптимального пути,
- 3) поскольку значение $g(v)$ в Q теперь стало меньше, обновляем приоритет очереди для v , показывая тем самым, что теперь от старта s до точки v можно добраться по более короткой траектории;

8. Возвращаемся к пункту 4.

Альтернатива Dijkstra при использовании NavFn Planner – это алгоритм A*, переключение на который осуществляется через такой параметр данного планировщика как `use_astar` (False по умолчанию).

С математической точки зрения, Navigation Function Planner с A* «под капотом» работает следующим образом:

1. Аналогично Dijkstra, создаётся граф $G=(V, E)$ (см. пункт 1 в предыдущем списке);
2. Вес w ребра между двумя соседними вершинами (клетками сетки) $u=(x_1, y_1)$ и $v=(x_2, y_2)$ задаётся как простое евклидово расстояние $d(u, v)$ между ними (1 – для прямых, $\sqrt{2}$ – для диагональных), без учёта штрафа $(1+\alpha * c(v))$, как это делается в Dijkstra, потому что если сильно увеличить вес ребра за счёт него, но не учесть этого в эвристике $h(v)$ (см. пункт 3), то она может перестать обладать двумя ключевыми свойствами – допустимостью и монотонностью, первое из которых гарантирует, что A* найдёт оптимальный путь за счёт того, что эвристика никогда не будет переоценивать реальную минимальную стоимость пути $h_{truth}(v)$ от текущей вершины v до цели:

$$h(v) \leq h_{truth}(v) \quad (23)$$

, пропуская таким образом лучшие решения из-за их неоправданно завышенной оценки.

Монотонность же в свою очередь гарантирует, что для каждого ребра (u, v) графа G , эвристика $h(u)$ будет удовлетворять следующему условию:

$$h(u) \leq c(u, v) + h(v) \quad (24)$$

где: $c(u, v)$ – равная весу ребра стоимость перехода из текущей ячейки u в соседнюю к ней проходимую ячейку v ; $h(v)$ – эвристическая оценка от вершины v до цели (см. пункт 3).

Формула (24) означает, что переход из u в v и дальнейшее движение от неё должно быть дороже по стоимости, чем двигаться напрямую с точки зрения эвристики – это и есть монотонность, которая предотвращает уменьшение приоритета $f = g + h$ (см. пункт 3) при перемещении по графу и, как следствие, исключает повторное рассмотрение его предыдущих узлов.

Если же всё-таки попытаться учесть этот штраф $(1 + \alpha * c(v))$ при расчёте веса w ребра, то скорректировать эвристику $h(v)$ будет довольно сложно, что может привести к потере оптимальности всего алгоритма A* в целом. Если же оставить вес ребра равным чистому расстоянию $d(u, v)$, не учитывая штраф, то эвристика будет правильной, но планировщик при этом может начать приближаться к потенциально опасным зонам;

3. Как и в случае очереди Q для Dijkstra, формируется приоритетная очередь O , которая содержит в себе значения так называемой функции приоритета $f(v)$ в порядке их возрастания:

$$f(v) = g(v) + h(v) \quad (25)$$

где: $g(v)$ в начале работы алгоритма формируется по тем же правилам, что и в Dijkstra (см. пункт 3 в предыдущем списке); $h(v)$ – эвристическая функция, которая позволяет оценить стоимость перехода от ячейки v до целевой точки пути g по следующим основным метрикам:

$$h(v) = \sqrt{(x_v - x_g)^2 + (y_v - y_g)^2} \quad (26) \text{ — евклидова}$$

$$h(v) = |x_v - x_g| + |y_v - y_g| \quad (27) \text{ — манхэттенская}$$

$$h(v) = \max(|x_v - x_g|, |y_v - y_g|) \quad (28) \text{ — диагональная}$$

4. Оставшиеся пункты 4-8 работы алгоритма A* будут отличаться от аналогичных им пунктов 4-8 работы алгоритма Dijkstra только тем, что работа будет вестись не с Q , а с O , и новая стоимость gn перехода из u в v будет рассчитываться не по (22), а по следующей формуле:

$$gn = g(u) + w(u, v) \quad (29)$$

где: $g(u)$ — накопленная стоимость перехода от старта s до текущей ячейки u с учётом всех предыдущих шагов; $w(u, v) = d(u, v)$ — стоимость перехода из текущей ячейки u в соседнюю к ней проходимую ячейку v **БЕЗ** учёта стоимости второй из Costmap (не путать со стоимостью ячейки и с накопленной стоимостью всего пути!).

Какой из двух описанных выше алгоритмов выбрать зависит от: 1) размерности сетки (Dijkstra обходит карту полностью), 2) её сложности (эвристика A* в виде евклидового расстояния может ошибочно проигнорировать нужный нам путь) и, как следствие, 3) производительности («звёздочка» быстрее).

Regulated Pure Pursuit

Regulated Pure Pursuit (далее – *RPP*) [37] является «регулируемой» в плане линейных скоростей версией исходного локального планировщика Pure Pursuit, устроенным следующим образом:

1. На вход подаётся список точек пути P , представленный их двумерными координатами X, Y и полученный от NFP;
2. Определяется ближайшая к текущему месторасположению робота точка p_r на траектории (см. Рис. 18), а от неё так называемая точка упреждения (англ. lookahead point) p_l по следующим формулам:

$$dist(p_i) = \sqrt{(x_r - x_i)^2 + (y_r - y_i)^2} \quad (30)$$

$$p_l = p_i \in P, \begin{cases} dist(p_{i-1}) < L \\ dist(p_i) \geq L \end{cases} \quad (31)$$

где: L – дистанция упреждения (англ. lookahead distance) между двумя точками, задаваемая в параметрах алгоритма;

3. Преобразовав координаты точки упреждения в систему координат робота, вычисляем кривизну траектории k как:

$$k = \frac{2 y_l'}{L^2} \quad (32)$$

где: y_l' – поперечная координата в системе координат робота;

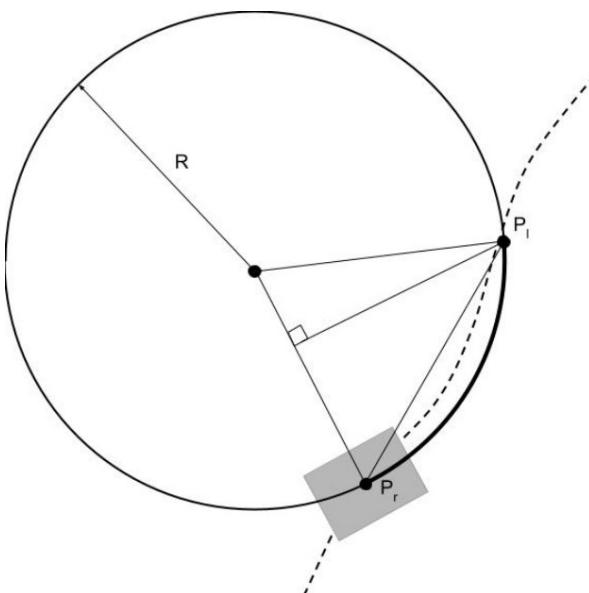


Рис. 18. Геометрия нахождения кривизны траектории

4. Рассчитывается управляющее воздействие в виде линейной скорости v_t как максимум из двух следующих эвристик:

$$v_t' = \begin{cases} v_t & \text{при } k > T_k, \\ \frac{v_t}{r_{\min} k} & \text{при } k \leq T_k \end{cases} \quad (33) - \text{эвристика кривизны (англ. curvature)}$$

$$v_t' = \begin{cases} v_t \frac{\alpha d_o}{d_{prox}} & \text{при } d_o \leq d_{prox}, \\ v_t & \text{при } d_o > d_{prox} \end{cases} \quad (34) - \text{эвристика сближения (англ. proximity)}$$

где: r_{\min} – минимально допустимый для робота радиус поворота; T_k – минимальный порог кривизны траектории, с которого начинает применяться первая эвристика; $\alpha \leq 1.0$ – множитель усиления второй эвристической функции (чем он выше, тем быстрее робот снижает свою скорость вблизи препятствий); d_o – текущее расстояние до препятствия; d_{prox} – минимальное расстояние до препятствий, с которого начинает применяться вторая эвристика;

5. Рассчитывается управляющее воздействие в виде угловой скорости ω_t по следующей формуле:

$$\omega_t = v_t' k \quad (35)$$

Что оригинальный алгоритм, что его разновидности могут использоваться для следования локальному пути роботами со всенаправленным (англ. omnidirectional) движением, однако они будут ограничены в выполнении боковых перемещений; роботами с дифференциальным (англ. differential) приводом, которые могут следовать по любой голономной (свободной от кинематических ограничений) траектории, но что самое главное, RPP может использоваться для следования локальному пути роботами с кинематикой Акерманна (автомобильной кинематикой), что было подтверждено на практике при тестировании решения (см. Рис. 56-58).

Model Predictive Path Integral

Model Predictive Path Integral (далее – *MPPI*) [38] является преемником алгоритма *TEB* [39], а также *MPC* [40], и так называемым локальным контроллером, работающим на основе предсказания и оптимизации движения в режиме реального времени, а также точного следования маршруту с учётом препятствий и кинематических ограничений.

Так же, как и для *Regulated Pure Pursuit*, глобальную траекторию для *MPPI* строит *NavFn Planner*, в то время как локальный планировщик работает в пределах заданного окна вокруг текущего положения робота следующим образом:

1. Генерируется набор управляющих воздействий (линейные и угловые скорости) путём добавления к ним случайного шума из гауссовского распределения. Каждая пара таких скоростей – это последовательность действий алгоритма длиной в T шагов вперёд (так называемый горизонт планирования, который задаётся такими параметрами данного планировщика, как `time_steps` и `model_dt`);
2. Эти действия прогоняются через динамическую модель робота с учётом его кинематики, на выходе выдавая множество траекторий движения;
3. Каждая из полученных траекторий оценивается по следующей формуле:

$$S_i = \sum_{t=0}^T c(x_t^i, u_t^i) \quad (36)$$

где: S_i – стоимость i -ой траектории; x_t^i – состояние на шаге t для траектории i ; u_t^i – управляющее воздействие на этом шаге; c – один из множества так называемых критиков или функций стоимости, отвечающих за учёт кинематики и препятствий, следование цели, отклонение от пути и т. д.;

4. Лучшее управляющее воздействие из всех отбирается с использованием функции softmax следующим образом:

$$\delta u = \frac{\sum_{i=1}^N \exp\left(-\frac{1}{\lambda} S_i\right) * \epsilon_i}{\sum_{i=1}^N \exp\left(-\frac{1}{\lambda} S_i\right)} \quad (37)$$

где: δu – величина корректировки крайнего управляющего воздействия; ϵ_i – шум; λ – значение параметра temperature, регулирующего чувствительность алгоритма к стоимости (чем ближе к нулю, тем больше принимаются во внимание траектории с меньшей стоимостью);

5. Полученная корректировка используется на следующей итерации, влияя на уровень генерируемого шума и, как следствие, на очередной набор из линейных и угловых скоростей.

Помимо роботов с кинематикой Акерманна, MPPI также поддерживает всенаправленных и дифференциальных роботов, что задаётся таким параметром алгоритма как *motion_model*.

Основной недостаток данного локального планировщика, который особенно сильно проявляется на более слабом железе, – это его высокая вычислительная сложность $O(N * T)$ из-за массового сэмплирования десятков, а то и сотен траекторий движения и их оценивания, где: N – количество траекторий, T – временной горизонт планирования. Для сравнения, сложность RPP – $O(1)$ из-за минимального объёма логики управления, отсутствия оптимизации и моделирования движения, а также простой структуры эвристик.

3 Тестирование решения

Одним из важнейших этапов разработки любого программного обеспечения является тестирование, которое, как правило, направлено на анализ его соответствия установленным требованиям (см. подразделы 1.4.2 «Функциональные требования» и 1.4.3 «Нефункциональные требования» главы 1) в целях обеспечения должного уровня качества ПО и его «понятности» для пользователя.

За счёт выполнения четвёртого нефункционального требования из списка, обозначенного выше, а именно – модульности, тестирование решения удобно будет производить именно по модулям, в том же логическом порядке, в котором они были описаны в предыдущей главе 2 «Проектирование и реализация решения».

Связка Webots + ROS 2 позволяет выбрать, какую конкретно сцену необходимо будет открыть в Webots при очередном запуске симуляции, а также легко переключаться между несколькими сценами путём указания названия одной из них в так называемом launch-файле, который позволяет запустить всё решение сразу выполнением всего одной консольной команды.

В данном контексте, сцена или мир – это основная единица моделирования, которая представляет из себя полноценное объектное представление виртуальной среды, включая физику, объекты окружения и роботов, а также их визуализацию (графику) и программное поведение.

Каждая такая сцена создаётся вручную с использованием графического пользовательского интерфейса в Webots и предназначается для выполнения определённого круга задач, отладки конкретных алгоритмов и тестирования заранее продуманных сценариев работы решения.

3.1 Калибровка виртуальных видеокамер

3.1.1 Эксперимент №1

Данный эксперимент нацелен на то, чтобы визуально попытаться оценить итоговый результат работы модуля «Calibration Tool» на примере эмуляции реальной модели видеокамеры и её последующей калибровки.

На Рисунке 19 ниже представлена сцена для калибровки виртуальных видеокамер. В левой части изображения можно наблюдать дерево характеристик (горизонтальный угол обзора, ширина и высота изображения, искажения объектива, фокусное расстояние, оптический центр и др.) узла под названием «camera_front_left», который представляет собой объект стереокамеры, расположенной в левой части 3D-модели беспилотного автомобиля и из которого исходят три оси X, Y и Z (см. Рис. 19). Расположение оставшихся трёх камер, а также углы их обзора можно определить по усечённой форме камеры (англ. camera frustum) в виде пирамид.

Напротив, и в области обзора каждого из статичных устройств расположены шахматные доски, которые начинают двигаться относительно них при запуске симуляции. В это время видеокамеры фиксируют эти паттерны калибровки в различных положениях и на основе известных размеров и геометрических преобразований подбирают свои внутренние параметры (фокусное расстояние, оптический центр и описанные в разделе 2.2 коэффициенты дисторсий), а также учитывают их впоследствии при устранении искажений на новых изображениях. Так работает «Новая гибкая технология калибровки камеры» [41], теоретическая основа которой была предложена ещё в 1999 году китайско-американским учёным в области компьютерного зрения Цзэнцю Чжаном.

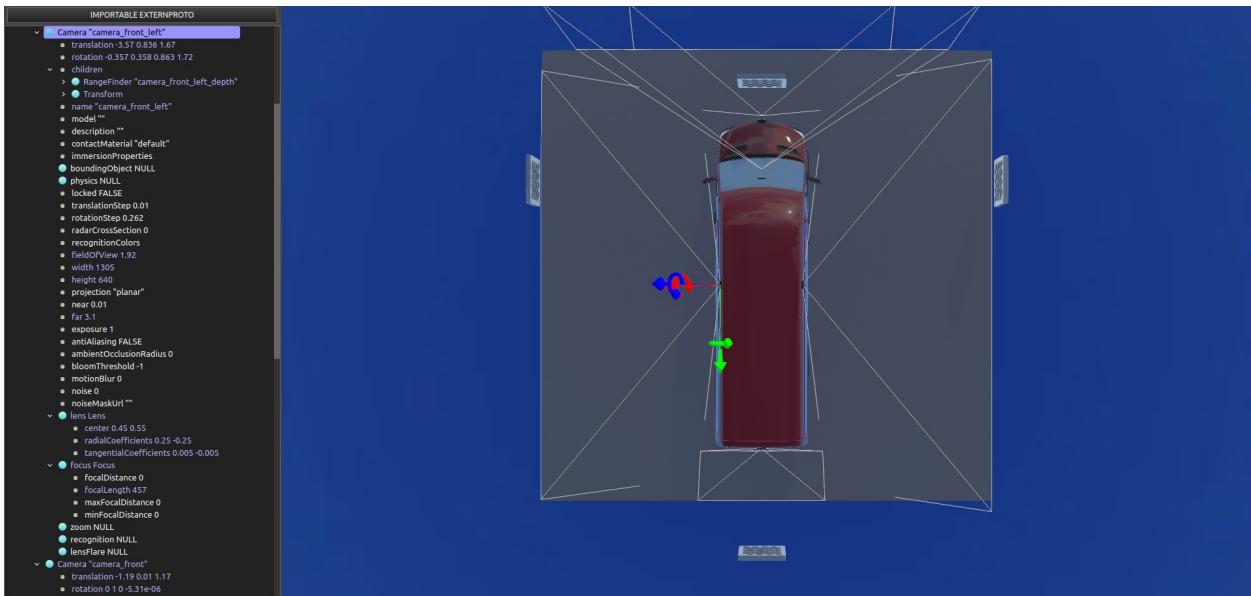


Рис. 19. Сцена №1 в Webots

Конкретная реальная модель камер, используемая в решении – ZED 2 от Stereolabs [42], которая обладает следующими ключевыми для данной работы характеристиками:

Таблица 2. Характеристики реальной стереокамеры модели ZED 2

Характеристика	Значение
Горизонтальный угол обзора	110°
Вертикальный угол обзора	70°
Выходное разрешение	2 x (1920x1080) @ 30 FPS 2 x (1280x720) @ 60 FPS
Фокусное расстояние	528 пикселей (HD720)
Минимальное значение глубины	0.3 м
Максимальное значение глубины	20 м

Для их симуляции и быстрого применения одновременно к нескольким узлам объектов видеокамер в Webots, был написан специальный Python-скрипт (см. Лист. 13 в **Приложении А** ниже) вместе с конфигурационным файлом для него, представленном в следующем Листинге 5:

```

# DEF-наименования объектов типа Camera в Webots
devices_name: [
    ...
    'camera_front_blind_depth',
    'camera_front_blind',
    ...
]

# Учитывать значения горизонтального и вертикального углов обзора
use_fov_values: True

horizontal_fov: 110 # Градусы
vertical_fov: 70 # 

image_width: 1305 # Пиксели

# Пересчитываются при use_fov_values: True
image_height: 640 # Пиксели
focal_length: 457 #

# Упрощённая модель искажения Брауна-Конради
distortion_center: [0.45, 0.55] # 0.0 ≤ x, y ≤ 1.0
radial_distortion: [0.25, -0.25] # k1, k2
tangential_distortion: [0.005, -0.005] # p1, p2

# Для стереокамер с объектом типа RangeFinder в Webots
min_range: 0.3 # Метры
max_range: 20.0 #

```

Листинг 5. Характеристики виртуальной стереокамеры модели ZED 2

Исполнение этого скрипта перезаписывает файл с описанием Webots-сцены, применяя таким образом указанные в конфиге параметры к узлам из списка devices_name (см. Лист. 5). Такой подход позволяет не только сократить время на задание всех этих параметров вручную для того, чтобы добиться желаемого результата, но и предоставляет возможность задать конкретную модель реальной камеры, которую планируется использовать, а также сымитировать описанные далее искажения её объектива и перспективы по «упрощённой» модели Брауна-Конради (см. раз. 2.2). Упрощённая она по тому, что в Webots (см. Рис. 20) отсутствует третий коэффициент k_3 радиальной дисторсии:

```

    • width 1305
    • height 640
    • projection "planar"
    • near 0.01
    • minRange 0.3
    • maxRange 20
    • motionBlur 0
    • noise 0
    • resolution -1
    ● lens NULL
  > ● Transform
  • name "camera_front_right"
  • model ""
  • description ""
  • contactMaterial "default"
  • immersionProperties
  ● boundingObject NULL
  ● physics NULL
  • locked FALSE
  • translationStep 0.01
  • rotationStep 0.262
  • radarCrossSection 0
  • recognitionColors
    • fieldOfView 1.92
    • width 1305
    • height 640
    • projection "planar"
    • near 0.01
    • far 0
    • exposure 1
    • antiAliasing FALSE
    • ambientOcclusionRadius 0
    • bloomThreshold -1
    • motionBlur 0
    • noise 0
    • noiseMaskUrl ""
  ● lens Lens
    • center 0.45 0.55
    • radialCoefficients 0.25 -0.25
    • tangentialCoefficients 0.005 -0.005
  ● focus Focus
    • focalDistance 0
    • focalLength 457
    • maxFocalDistance 0
    • minFocalDistance 0

```

Рис. 20. Применённые параметры правой виртуальной стереокамеры модели ZED 2

Перед запуском симуляционной среды (см. Рис. 19), последнее, что остаётся сделать – это вручную задать траектории движения для каждого из паттернов калибровки путём их физического перемещения на сцене №1 в симуляторе и копирования очередного набора значений координат X, Y и Z в соответствующий словарь (см. Лист. 6 ниже) из скрипта под названием «chessboard_controller_node.py». Эти траектории необходимы для того, чтобы каждый новый такт (шаг) симуляции получать немного видоизменённое калибровочное изображение шахматной доски для каждого из девайсов.

```

chessboards_movement_trajectory = {
    'chessboard_front_left': [
        ['X', 3.0],
        ['Z', [-0.476905, 0.621515, 0.621515, -2.25159]],
        ['Z', [-0.377964, 0.654654, 0.654654, -2.41886]],
        ...
    ],
    'chessboard_front': [
        ['Y', -3.25],
        ['X', 0.6],
        ['X', 0.1],
        ...
    ],
    'chessboard_front_blind': [
        ['Y', -3.25],
        ['X', 0.6],
        ['X', 0.1],
        ...
    ],
    'chessboard_front_right': [
        ['X', -3.0],
        ['Z', [-0.677661, -0.519988, -0.519988, -1.95044]],
        ['Z', [-0.774596, -0.447214, -0.447214, -1.82348]],
        ...
    ],
    'chessboard_rear': [
        ['Y', 4.75],
        ['X', -0.4],
        ['X', 0.1],
        ...
    ],
}

```

Листинг 6. Словарь, содержащий в себе траекторию движения для каждой из шахматных досок

Выполнение всех описанных выше действий приводит к тому, что атрибуты объектов на сцене №1 в Webots принимают (см. Рис. 20), или будут принимать при запуске симуляции, как в случае движения шахматных досок, значения, указанные в `webots_settings.yaml` (см. Лист. 5) и в `chessboards_movement_trajectory` (см. Лист 6).

По итогу, конечным результатом работы модуля «Calibration Tool» является выходной конфигурационный `.yaml`-файл (см. Лист. 7) для каждой из видеокамер, содержащий в себе подобранную матрицу K её внутренних параметров, таких как фокусное расстояние и оптический центр по осям X и Y , а также матрицу D , состоящую из двух коэффициентов радиальной и из двух коэффициентов тангенциальной дисторсий:

```
%YAML:1.0
---
image_resolution: !!opencv-matrix
...
data: [ 640, 1305, 4 ] # image_height, image_width, channels (depth)
camera_matrix: !!opencv-matrix
...
# fx, 0., cx, 0., fy, cy, ...
data: [ 456.88983054030831, 0., 652., 0., 456.88983054030831, 319.5,
0., 0., 1. ]
distortion_coefficients: !!opencv-matrix
...
# k1, k2, p1, p2, k3
data: [ 0.09343090353266352, -0.030199853852743441,
-0.0027583853029400809, 0.0047953056439332372, 0. ]
projection_matrix: !!opencv-matrix
...
# Коэффициенты гомографии h11-h33:
# h11, h22 – масштаб; h12, h21 – скос; h13, h23 – смещение; h31, h32 – изгиб; h33 – нормализация
data: [ 0.74478089148015814, 2.8458247131841286, -91.595824713184186,
-4.7085718174959067e-18, 3.0186195222870049, -37.95373330825651,
-2.0990932906778326e-20, 0.0071939063381606227, 1. ]
```

Листинг 7. Подобранные алгоритмами OpenCV параметры одной из камер

Применив их для устранения искажений, получим следующее результирующее изображение:

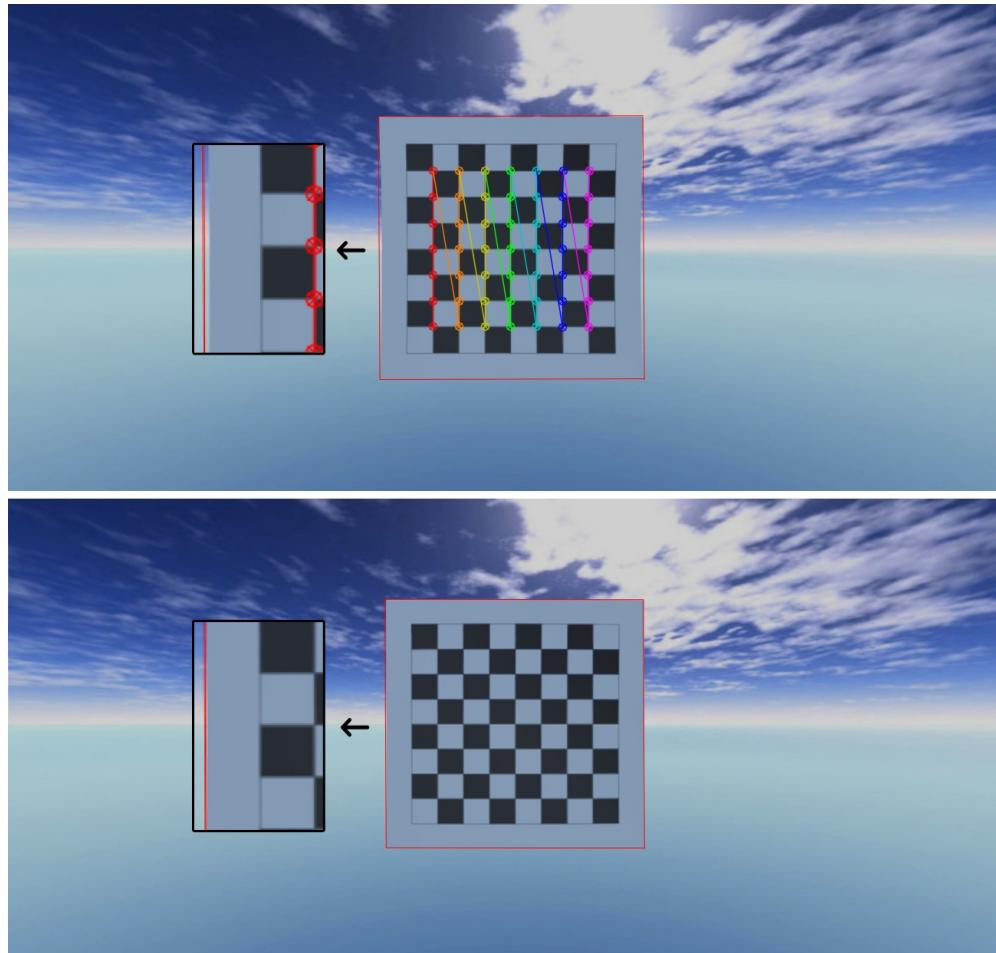


Рис. 21. Искажённое (сверху) и результирующее (снизу) изображения

Таким образом, данный эксперимент показал, что визуальный итоговый результат (см. Рис. 21) работы модуля «Calibration Tool» в целом соответствует ожиданиям о том, что в действительности должно получаться на выходе процесса калибровки уже реальных видеокамер – матрицы их параметров K и D , применение которых к новым изображениям сделает фактически прямые линии на них параллельными, а также позволит устраниТЬ перспективные искажения, если таковые имеются.

3.1.2 Эксперимент №2

Данный эксперимент нацелен на то, чтобы через выполнение трёх серий измерений математически оценить точность соответствия моделей искажения, применяемых в Webots и в OpenCV, что говорится по «сухим цифрам», и на всё том же примере, что был составлен в первом эксперименте.

Выше был описан процесс калибровки виртуальных видеокамер в целом. Теперь же оценим схожесть эмулированных в симуляторе значений оптических свойств, таких как коэффициенты радиальной дисторсии k_1 и k_2 , и тангенциальной дисторсии p_1 и p_2 (см. правую часть на Рис. 20 выше), с подобранными алгоритмом «Новой гибкой технологии калибровки камеры» из библиотеки компьютерного зрения (см. Лист. 7 выше).

Порядок выполнения одной серии измерений практически полностью соответствует порядку проведения Эксперимента №1 из подраздела 3.1.1, за исключением того, что такие параметры из Листинга 5, как `radial_distortion` и `tangential_distortion`, отвечающие за значения обозначенных ранее коэффициентов в симуляторе, будут для каждой новой серии принимать разные величины – от идеальных условий без каких-либо искажений, до более сложных, где они уже могут быть ощутимы. Такие изменения в Листинге 5 будут закономерно сказываться на том, что по итогу будет подобрано алгоритмом и сохранено в конфиг, в формате Листинга 7.

На Рисунках 22-24 ниже приведено сравнение искусственных коэффициентов радиальной дисторсии k_1 и k_2 , и тангенциальной дисторсии p_1 и p_2 , заданных в Webots, с теми, что подобрали алгоритмы OpenCV:

```

radial_distortion: [0.0, 0.0] # k1, k2
tangential_distortion: [0.0, 0.0] # p1, p2
distortion_coefficients: !!opencv-matrix
  rows: 5
  cols: 1
  dt: d
  data: [ 0.00047439189029067924, -0.000678978953370386,
    | 2.8219612300705556e-05, -2.5558263627103442e-05, 0. ]

```

Рис. 22. Заданные в Webots (сверху) и подобранные OpenCV (снизу) коэффициенты №1

```

radial_distortion: [0.25, -0.25] # k1, k2
tangential_distortion: [0.005, -0.005] # p1, p2
distortion_coefficients: !!opencv-matrix
  rows: 5
  cols: 1
  dt: d
  data: [ 0.092066747091912413, -0.024912790710435662,
    | -0.0027641878976963161, 0.0044012073886357207, 0. ]

```

Рис. 23. Заданные в Webots (сверху) и подобранные OpenCV (снизу) коэффициенты №2

```

radial_distortion: [-0.5, 0.5] # k1, k2
tangential_distortion: [-0.01, 0.01] # p1,
distortion_coefficients: !!opencv-matrix
  rows: 5
  cols: 1
  dt: d
  data: [ -0.17457933435580689, 0.053093358340775276,
    | 0.0045108743933011631, -0.0057687900633389385, 0. ]

```

Рис. 24. Заданные в Webots (сверху) и подобранные OpenCV (снизу) коэффициенты №3

Таким образом, модели искажения, применяемые в Webots и в OpenCV практически не соответствуют друг другу, поскольку в робототехническом симуляторе используется «упрощённая» модель искажения Брауна-Конради без третьего коэффициента p_3 тангенциальной дисторсии, который в полной модели, вероятнее всего используемой в OpenCV, пришлось обнулить.

Несмотря на это, предыдущие визуальные результаты на Рисунке 21 выше, дают некоторые основания полагать, что внутренняя взаимосвязь между моделями искажения всё-таки присутствует, пусть и не один к одному.

Более высокие значения задаваемых в Webots искажений, чем те, что перечислены выше, нецелесообразны, так как явно указывают на очень сильное смещение линзы и необходимость её замены из-за дефекта при использования реальной видеоаппаратуры или же искусственного понижения значений коэффициентов до более реальных при работе в симуляторе.

```
focal_length: 457 #  
  
# Brown-Conrady distortion model  
distortion_center: [0.45, 0.55] # 0.0 ≤ x, y ≤ 1.0  
camera_matrix: !!opencv-matrix  
    rows: 3  
    cols: 3  
    dt: d  
    data: [ 455.40140577414064, 0., 652., 0., 455.40140577414064, 319.5,  
           | 0., 0., 1. ]
```

Рис. 25. Значения фокусного расстояния и оптического центра в Webots (сверху) и OpenCV (снизу)

Таким образом, данный эксперимент показал, что несмотря на низкую степень соответствия моделей искажения, применяемых в Webots и в OpenCV, полученные значения ошибок перепроектирования (0.016, 0.049, 0.016 и 0.036) свидетельствуют о достаточной точности найденных в процессе калибровки параметров для левой, передней, правой и задней камер соответственно. Скорее всего, это связано с тем, что часть внутренних параметров, которые были подобраны довольно точно во всех трёх сериях измерений (см. Рис. 25), компенсируют сильное расхождение в другой их части, обоснованное ранее.

3.2 Система двухмерного кругового обзора

3.2.1 Эксперимент №1

Данный эксперимент нацелен на то, чтобы правильно настроить для дальнейшей работы и практического применения модуль «Surround View», используя для этого представленную на Рисунке 26 ниже сцену для формирования схожего с этим видом на неё сверху единого изображения, но делая это не просто путём выставления камеры сцены в симуляторе, как на всё том же Рисунке 26, а алгоритмически, за счёт объединения изображений с пяти видеокамер, установленных по периметру БНТС (см. Рис. 19).

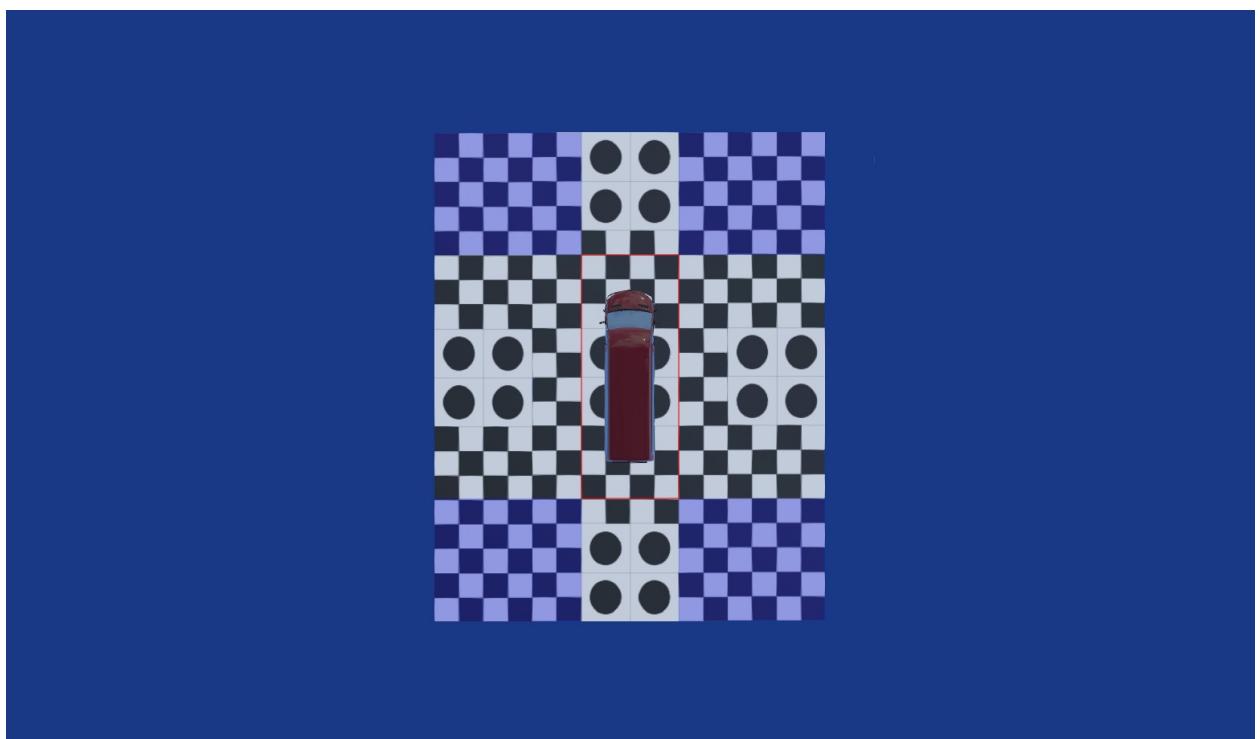


Рис. 26. Сцена №2 в Webots

Виртуальные размеры напольной шахматной доски – 20x25 метров, её одной клетки – 1.25 метра, квадрата с четырьмя окружностями внутри – 5 метров.

Теперь, после того как виртуальная среда проведения данного эксперимента была рассмотрена, перейдём к поэтапному процессу взаимодействия с системой двухмерного кругового обзора:

1. Запускаем узел `projection_weight_matrices_node` (см. Рис. 5) через исполнение launch-файла под названием «`get_projection_weight_matrices_launch`»;
2. Перспективные изображения (см. Рис. 27) в виде байт начинают поступать с пяти проинициализированных при запуске объектами класса `CameraModel` (см. Рис. 4) камер из Webots и преобразовываться в NumPy-массивы, которые затем передаются в метод `get_projection_matrix`:

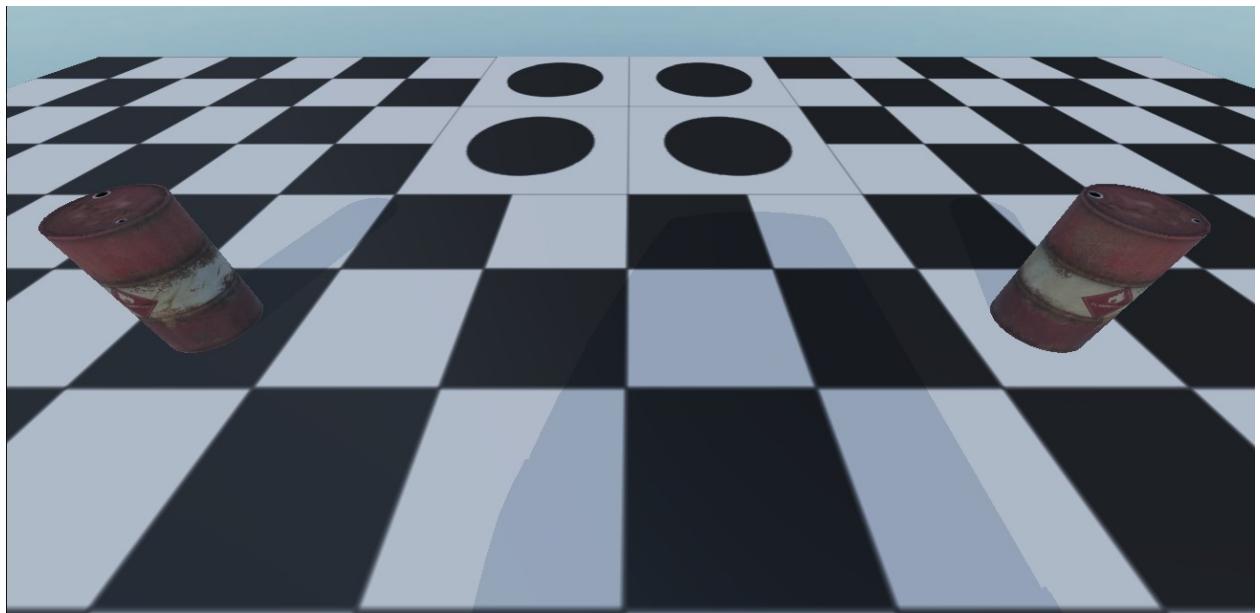


Рис. 27. Перспективное изображение с задней видеокамеры

3. Вызов этого метода запускает инструмент `PointSelector`, который позволяет отметить на исходном изображении четыре точки, составляющие область проекции (см. Рис. 28), по которой впоследствии будет рассчитана проекционная матрица, как это было описано в пункте 18 списка из раздела 2.2 «**Калибровка виртуальных видеокамер**» главы 2. Отдельно стоит обратить внимание на то, что параметры кругового обзора из Листинга 1 изначально подбирались эмпирическим путём и их дополнительный подбор либо же уточнение, с целью повышения итогового качества, не будут производиться в рамках данного и последующего эксперимента в связи с трудоёмкостью данного процесса:

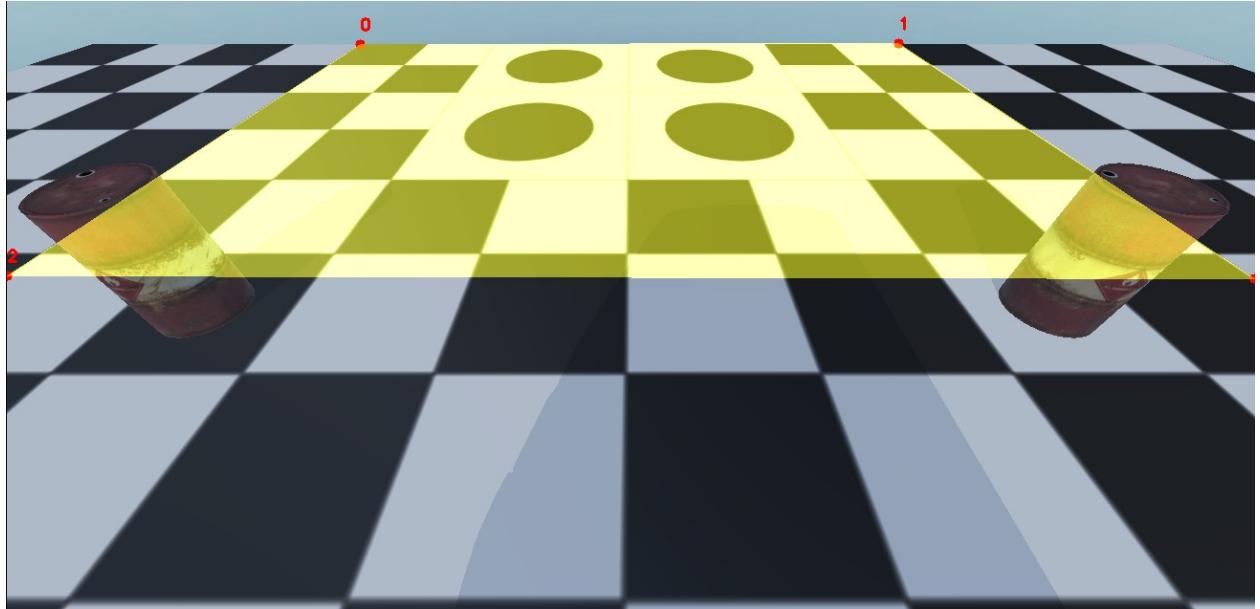


Рис. 28. Выделенная жёлтым через PointSelector область проекции

После того, как точки были отмечены строго в этом же порядке, нажимаем клавишу «Enter», чтобы рассчитать и сохранить в файл матрицу проекции по выделенной на перспективном изображении области с учётом заданных параметров кругового обзора (см. Лист. 1). Получаем следующее изображение с видом сверху:

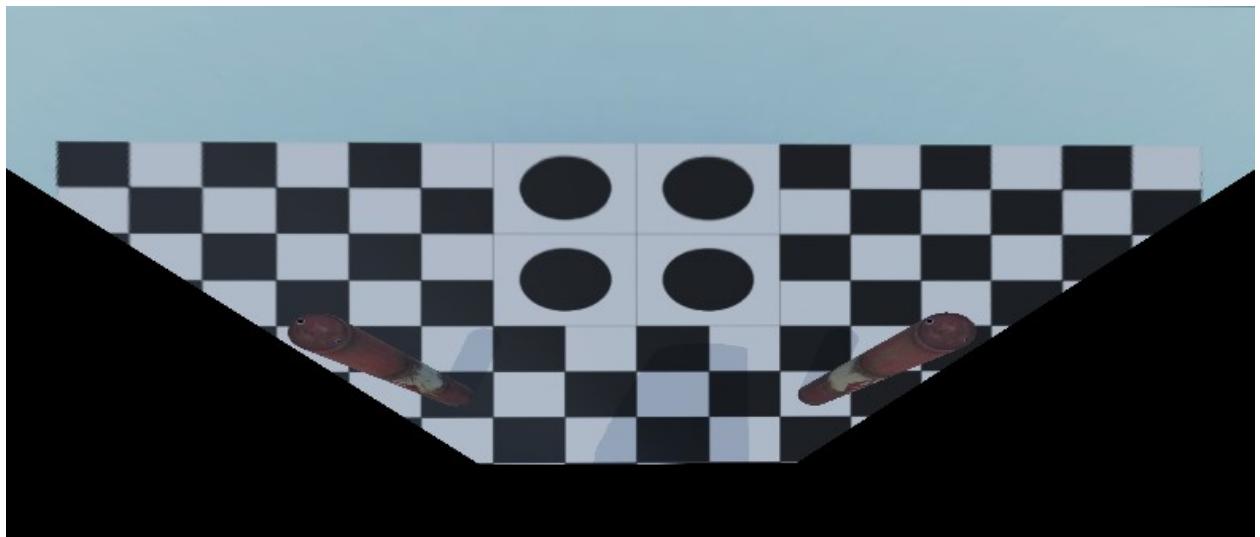


Рис. 29. BEV-изображение с задней видеокамеры

На данном этапе крайне важным является область проекции, которая была выделена, и масштаб *BEV-изображения*, который получится в результате применения гомографии. Сам он зависит как от фактического расположения устройства относительно выделенной области и её размеров, так и от упомянутых ранее параметров, изменение которых влияет на работу сразу всей системы двухмерного кругового обзора.

Все эти факторы, а также ещё и тот, что масштаб должен быть примерно, если не точно, одинаковым для всех сшиваемых BEV-изображений, и делают данный процесс трудоёмким и нетривиальным. Если же попытаться ими пренебречь, это может привести к результату, представленному на Рисунках 30-31 ниже, а также к тем проблемам, которые были выявлены при проведении **Эксперимента №2** далее:

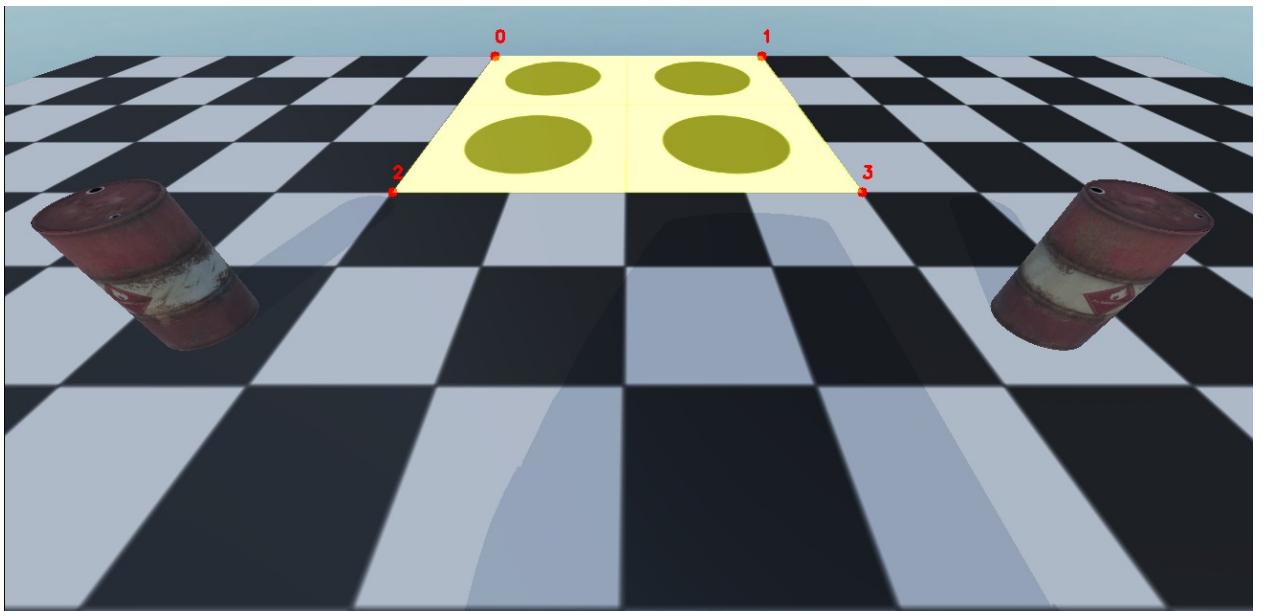


Рис. 30. Выделенная жёлтым через PointSelector неадекватная область проекции

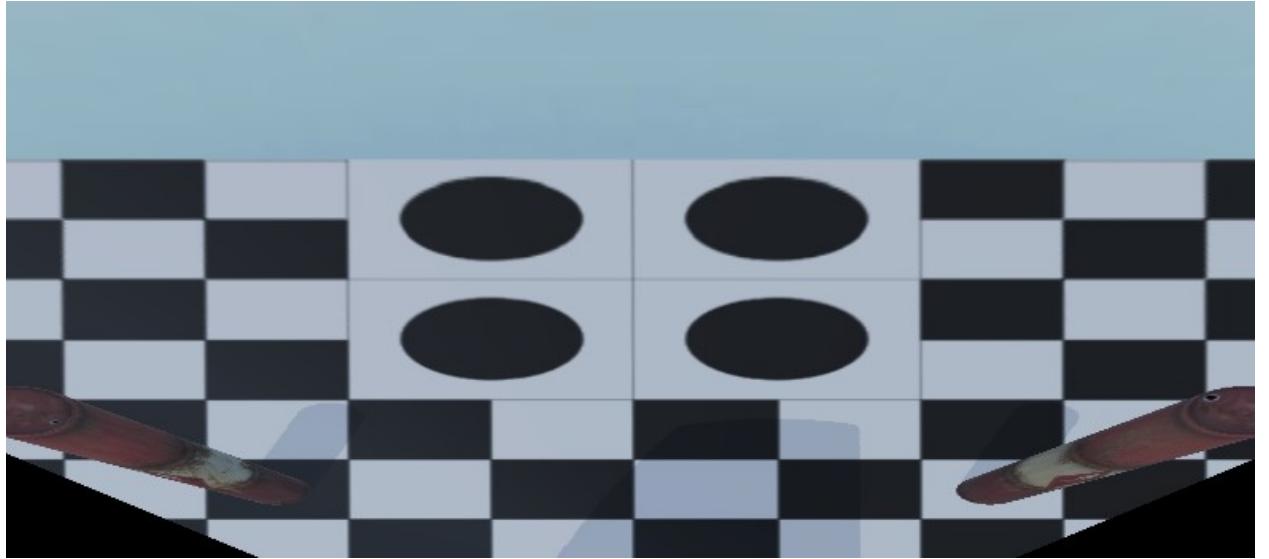


Рис. 31. Соответствующее ей BEV-изображение

4. Предыдущий этап проделывается с каждой из используемых камер, что на выходе даёт пять матриц проекции (см. Лист. 7), которые применяются к исходным изображениям, сохраняются в словарь `images_projected` всё в том же узле `projection_weight_matrices_node`, который в таком виде затем передаётся в качестве аргумента в метод `get_weight_matrix`, запуская таким образом цепочку шагов 3-6 списка из раздела 2.3 «**Система двухмерного кругового обзора**» главы 2;
5. Итоговый результат работы данного метода и системы двухмерного кругового обзора в целом, представлен на Рисунке 32 ниже, визуальный эффект вытягивания высоких объектов на котором, например, бочек, возникает от того, что применяемое в системе гомографическое преобразование корректно работает только для низких объектов, лежащих на одной плоскости с нулевым уровнем (землёй), в то время как высокие объекты выступают над этой плоскостью. При проецировании такая высота не учитывается и объекты как бы отбрасываются на землю, но со смещением от реальной точки основания:

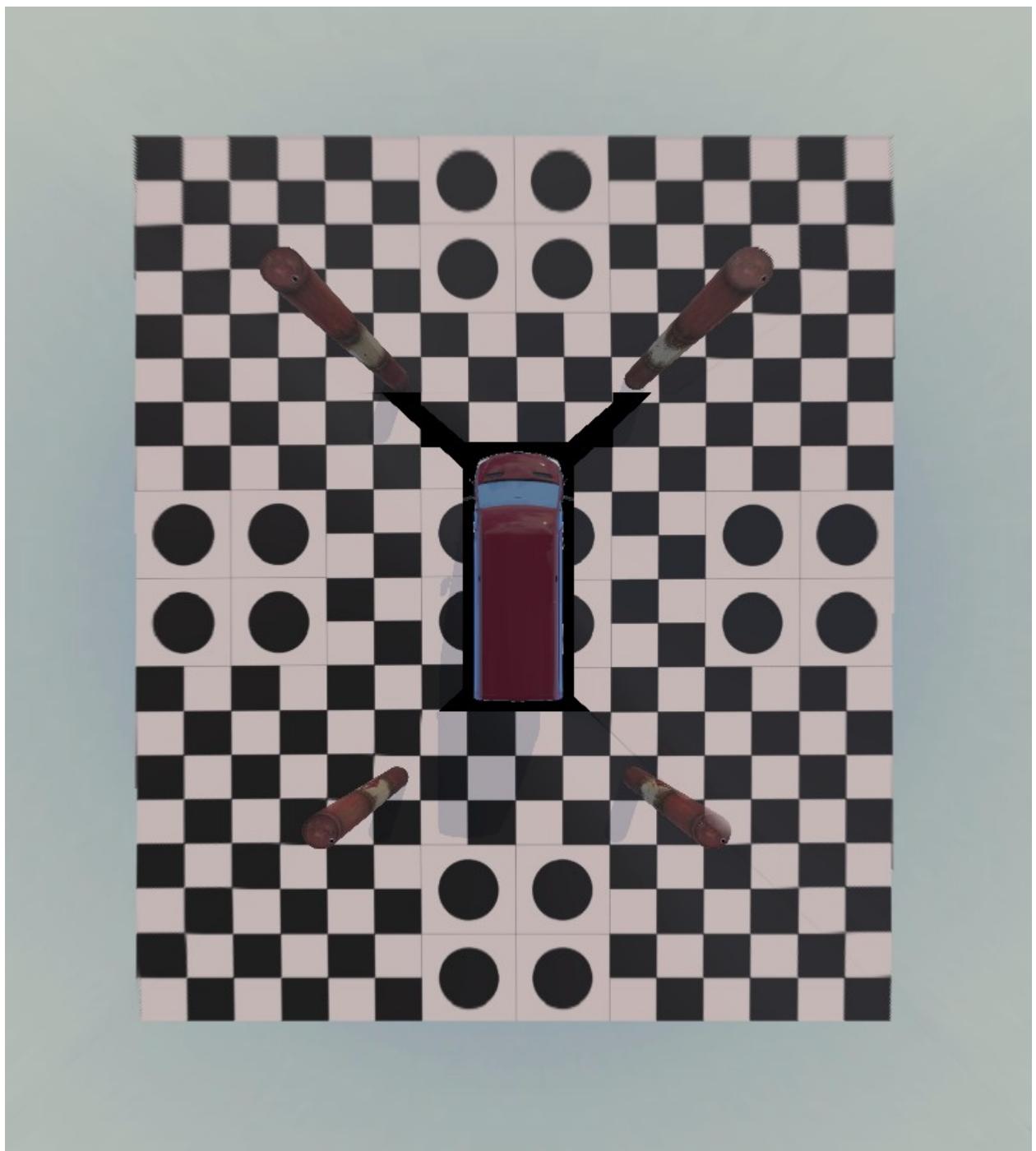


Рис. 32. Итоговый результат работы системы двухмерного кругового обзора

В случае же, если перечисленные в пункте 3 факторы не были учтены в полной мере, то и итоговый результат вследствие этого может получиться следующим:

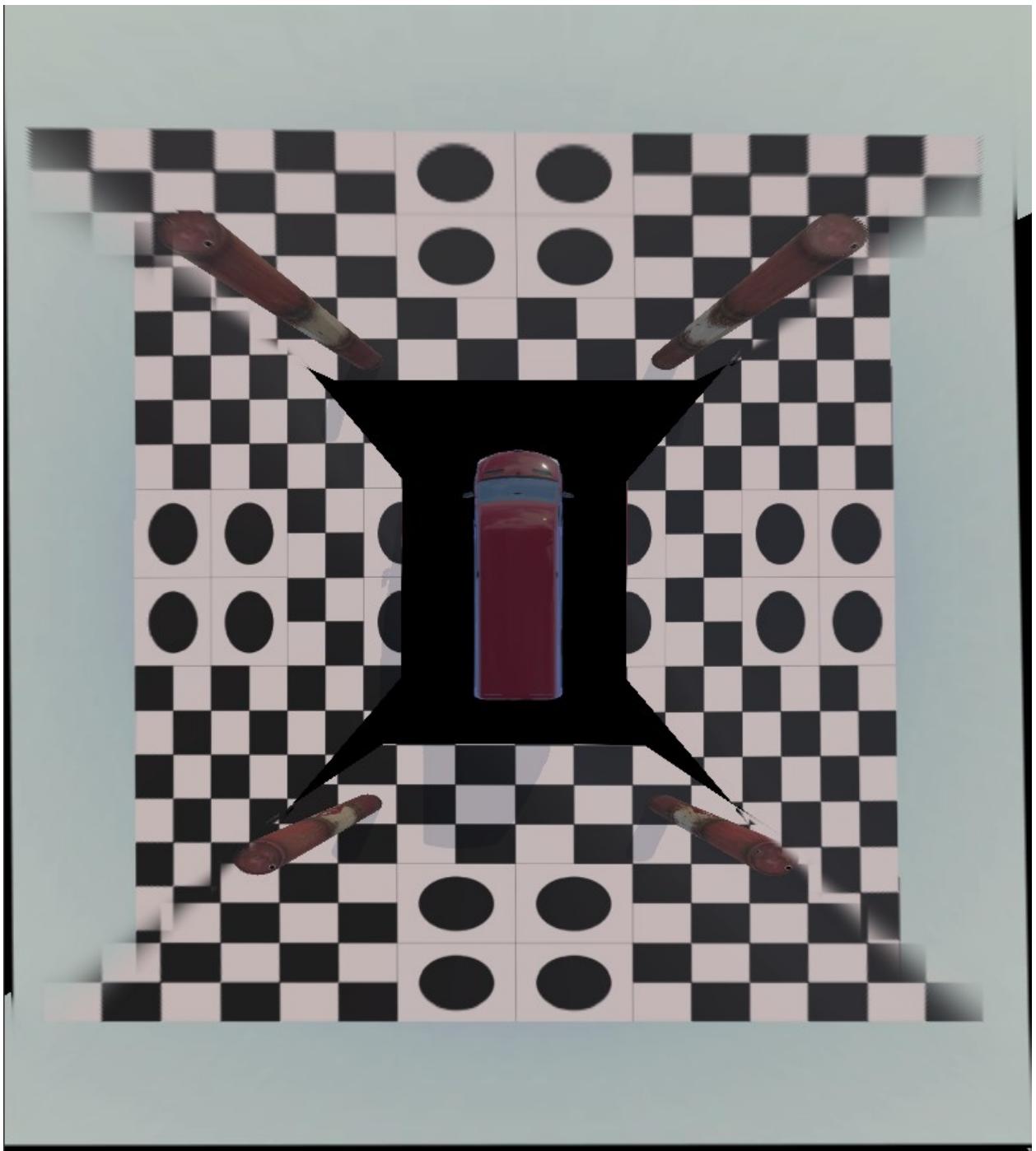


Рис. 33. Искажённый итоговый результат работы системы двухмерного кругового обзора

Таким образом, данный эксперимент показал, что получившаяся система двухмерного кругового обзора оказалась крайне чувствительна не только к правильному и симметричному расположению используемых виртуальных видеокамер вокруг эго-автомобиля в Webots, но и к расчёту их проекционных матриц по так называемой области проекции, выделяемой с использованием специально написанного для этого инструмента под названием «PointSelector» (см. Рис. 28, 30).

3.2.2 Эксперимент №2

Данный эксперимент нацелен на то, чтобы протестировать работу системы двухмерного кругового обзора в условиях, более приближенных к реальным, то есть – в движении и с разнообразным на объекты окружающим пространством. При описании сцены №2 для формирования системы двухмерного кругового обзора, на Рисунке 32 выше уже был представлен её итоговый результат работы, но это было сделано как бы «в вакууме», потому что сама по себе данная сцена (см. Рис. 26) очевидно не пригодна для дальнейшей отладки и тестирования других модулей решения.

На Рисунке 34 ниже представлена сцена с полосой препятствий, которая будет использоваться во всех оставшихся экспериментах.

Данный мир представляет из себя испытательный полигон с препятствиями в виде пластиковых бочек. Задача эго-автомобиля в режиме автономного управления доехать до конца полигона, развернуться с использованием задней передачи в свободной от препятствий зоне и вернуться обратно, попутно со всем этим обезжая бочки. Сигналы светофора и знаки дорожного движения при этом никак не учитываются, однако, в случае необходимости, могут быть использованы для проверки работоспособности и точности прочих алгоритмов.



Рис. 34. Сцена №3 в Webots

На Рисунках 35-38 представлены итоговые результаты тестирования работы системы двухмерного кругового обзора в более «боевых» условиях испытательного полигона сцены №3 в Webots:



Рис. 35. Вид сверху в симуляторе (слева) и итоговый двухмерный круговой обзор (справа)



Рис. 36. Большая часть передней бочки отсутствует из-за неидеального соединения (слева)



Рис. 37. Половина задней бочки отсутствует из-за неидеального соединения изображений

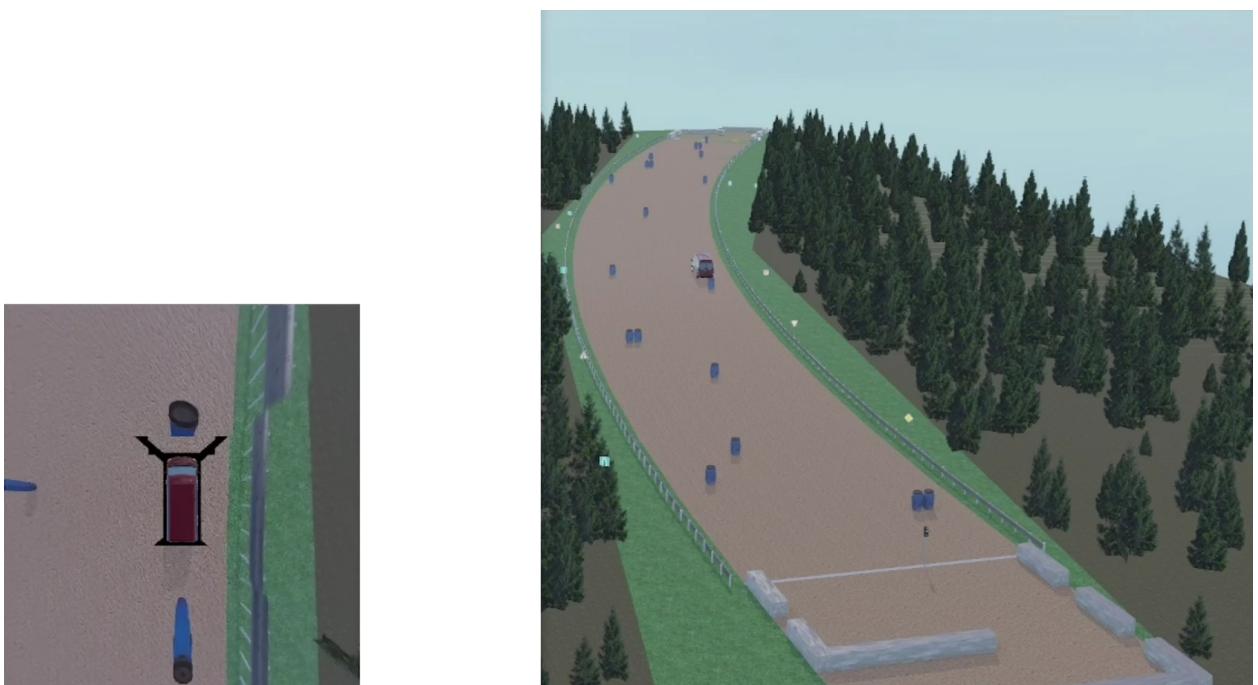


Рис. 38. Несоответствие масштабов на переднем, правом и заднем изображениях (слева)

Таким образом, данный эксперимент показал, что итоговый результат работы системы двухмерного кругового обзора в условиях, более приближенных к реальным, оказался не таким идеальным, как на Рисунке 32, поскольку в процессе тестирования системы было выявлено, что объекты на стыках сшиваемых изображений имеют свойство пропадать из поля видимости (см. Рис. 36-37) по причинам, описанным в **Эксперименте №1**.

Помимо этого, несоответствие масштабов боковых (правого и левого) изображений с передним и задним (см. Рис. 38) довольно сильно искажают геометрию объектов мира в Webots, что может привести к значительному уменьшению точности работы алгоритмов детекции и сегментации, а это, в свою очередь, напрямую повлияет на безопасность и эффективность автономного движения БНТС с использованием сегментированной локальной карты.

Чтобы избежать подобных ситуаций, требуется более точная донастройка BEV через подбор его параметров из Листинга 1 выше, что позволит не только выровнять масштаб между всеми четырьмя сшиваемыми изображениями и сделать его одинаковым, но также устранить «эффект пропажи» потенциально ценной для алгоритмов информации на их стыках.

3.3 Сегментированная локальная карта

3.3.1 Эксперимент №1

Данный эксперимент нацелен на то, чтобы оценить качество работы модуля «SegBEV» для осуществления по сегментированной локальной карте, которую он строит, планирования маршрута движения эго-автомобиля с точки зрения точности и производительности моделей сегментации и детекции, применяемых для формирования карты. Также, по ходу проведения данного эксперимента будет детально описан процесс их обучения.

Сам по себе, круговой обзор (см. Рис. 35) бесполезен, если речь идёт именно о беспилотном наземном транспортном средстве, потому что компьютер не может просто взять и поехать по нему, как это может сделать человек, – его нужно этому научить.

Во введении данной пояснительной записи было упомянуто, что «без надёжного и устойчивого алгоритма сегментации невозможно построение корректной сегментированной локальной карты на основе системы двухмерного кругового обзора, что, в свою очередь, затрудняет эффективное, а что самое главное безопасное планирование маршрута движения беспилотных наземных транспортных средств».

Использование такой модели глубокого обучения, как FastSeg, решает обе эти проблемы, поскольку свёрточные нейронные сети, каковой она и является, в задачах сегментации и детекции уже довольно давно превзошли классические подходы машинного обучения как по надёжности, так и по устойчивости [43].

На Рисунке 39 ниже представлен кадр из набора обучающих данных для этой модели, которые собирались путём ручного управления виртуальным БНТС в Webots с клавиатуры через использование узла ackermann_keyboard_teleop_node (см. Рис. 1) и фиксации с последующим сохранением в отдельные директории изображений со всех пяти видеокамер одновременно, что позволило значительно ускорить данный процесс.

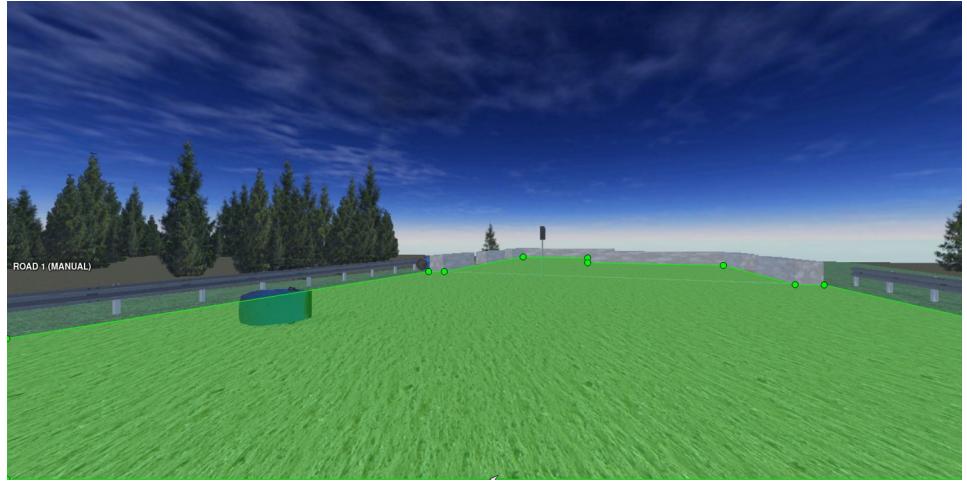


Рис. 39. Пример кадра из набора обучающих данных для модели FastSeg

Выделенная вручную область зелёного цвета – это так называемая маска сегментации, которая в веб-приложении *CVAT* [44], предназначенном для разметки собранных данных, имеет свой класс под названием «road» (англ. дорога), а также набор известных координат, по которым она может быть программно извлечена. Всё, что находится за пределами этого набора, дорогой не является и сеть, при её обучении, интересовать не должно.

Чтобы она это понимала, человеческая разметка, после завершения всего процесса в CVAT, дополнительно преобразуется в понятный для неё формат уже бинарной маски (см. Рис. 40), на которой белые пиксели со значением 1 формируют из себя размеченную ранее область проезда, пригодную для автомобиля, в то время как все остальные чёрные пиксели, находящиеся за пределами данной области, принимают значение 0.



Рис. 40. Кадр выше в формате бинарной маски

После того, как описанным выше образом *датасет* был полностью собран и размечен, а модель сегментации обучена с использованием бесплатных мощностей платформы Kaggle [45] и её сервиса под названием «Kaggle Notebook», можно наконец опробовать её в действии, что и было сделано на следующем Рисунке 41:

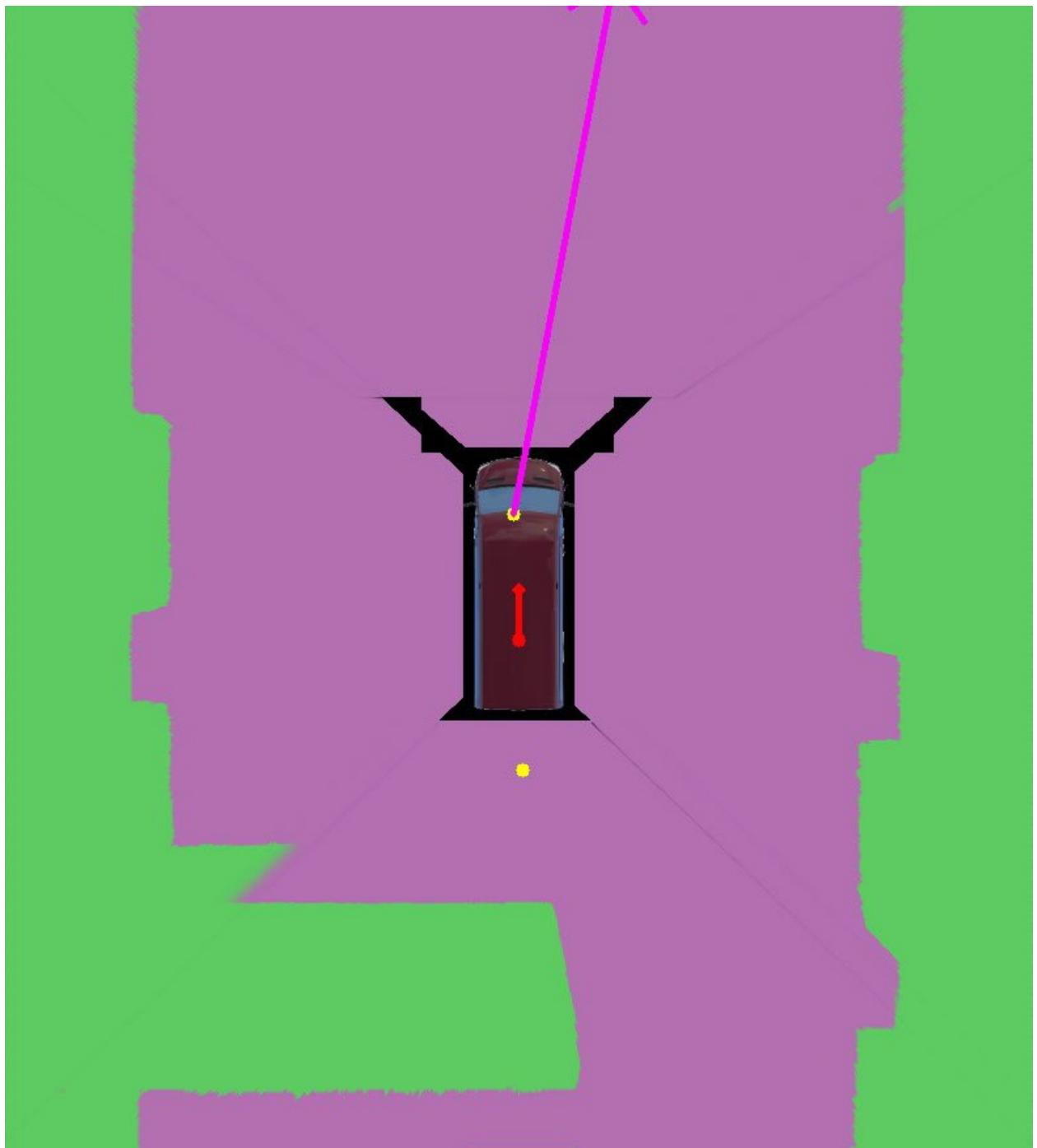


Рис. 41. Результат применения модели FastSeg к двухмерному круговому обзору

Поскольку свои предсказания FastSeg делает целых пять раз (по одному для каждой из используемых камер) для создания по ним всего одного изображения кругового обзора (см. Рис. 41), то это очень сильно снижает производительность всего решения в целом из-за фактически пятикратного увеличения довольно дорогостоящих вычислений.

Кроме этого, на всё том же Рисунке 41 ещё более наглядно видны некоторые из линий соединения пяти изображений в одно, что искажает общую картину восприятия и в дальнейшем потенциально может привести к некорректной работе алгоритмов планирования маршрута движения БНТС в более нестандартных и сложных ситуациях.

В связи со всем этим, было принято решение аналогичным образом собрать, разметить и обучить данную модель глубокого обучения на другом датасете, пример кадра из которого представлен на следующем Рисунке 42:

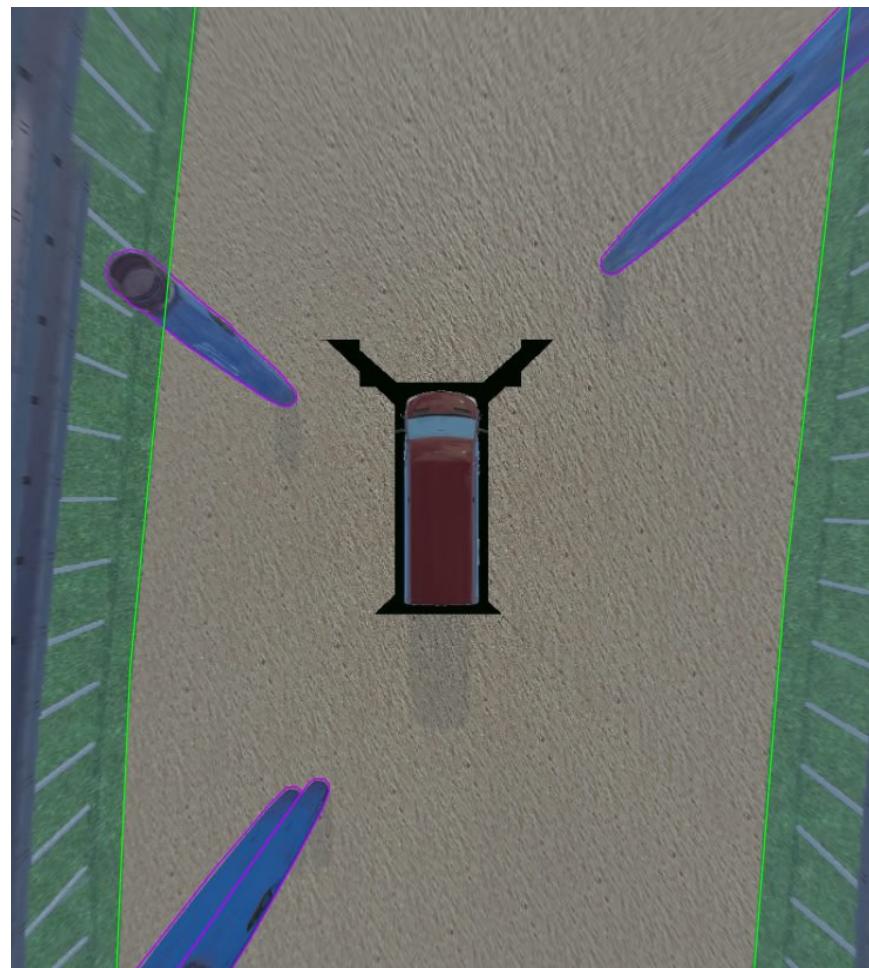


Рис. 42. Пример кадра из набора обучающих данных для модели FastSegBEV

FastSegBEV – это просто другое обозначение той же самой с точки зрения архитектуры модели FastSeg, за исключением того, что она делает свои предсказания не на пяти изображениях в отдельности, которые затем объединяются в одно, а уже на едином сшитом изображении кругового обзора, что позволяет не только получить сильный прирост в производительности, но и повысить общую точность сегментации за счёт того, что неровные стыки на исходном изображении будут перекрываться наложенной на него маской, предсказанной FastSegBEV, а не сначала делаться предсказания, как в случае с FastSeg, которые затем будут не совсем ровно склеиваться воедино.

Очевидный недостаток такого подхода – это специфичность данных, с которыми приходиться работать, а именно – BEV-изображения, уже собранных датасетов с которыми гораздо меньше в свободном доступе в сети Интернет, чем с теми же привычными перспективными изображениями, которые требуются для работы с FastSeg.

Несмотря на то, что FastSegBEV был дополнительно обучен сегментировать синие пластиковые бочки, представляющие из себя препятствия, это работает медленнее по сравнению с использованием отдельной и оптимизированной под такие задачи модели детекции, а также плохо поддаётся масштабированию на другие виды препятствий из-за дополнительных временных затрат на их отдельное выделение в процессе разметки собранных данных.

Пример кадра из набора обучающих данных для такой модели детекции, используемой в тестируемом решении, а именно – YOLO11, представлен на Рисунке 43 ниже:



Рис. 43. Пример кадра из набора обучающих данных для модели YOLO11

Сбор и обучение производились аналогично модели FastSeg, отличается только разметка, которая представляет из себя не маску, а так называемый ограничивающий прямоугольник или рамку с набором известных координат (см. Рис. 43), которые в виде обычного текстового файла с их перечислением, совместно с исходным изображением, подаются на вход свёрточной нейронной сети для её обучения:

0 0.267548 0.107891 0.086360 0.215781

0 0.573180 0.113359 0.060843 0.131719

, где: 0 – класс «plastic_barrel» ([англ.](#) пластиковая бочка); 0.267548 0.107891 ... – нормализованные координаты ограничивающего прямоугольника на исходном изображении.

Помимо уже частично решённых проблем надёжности и устойчивости алгоритмов сегментации и детекции, существует и другая насущная проблема, с которой учёные и разработчики борются до сих пор – данные, их качество, а с недавних пор ещё и количество [46].

В этой работе данную проблему было решено обойти стороной и собрать небольшое количество (100 изображений с каждой из используемых стереокамер) только тех данных, которые обе модели будут «видеть» на постоянной основе.

Иными словами, FastSeg и YOLO11 были намеренно переобучены и делали свои предсказания (см. Рис. 44) только на тех изображениях, на которых и обучались. Такой подход позволил сосредоточиться на разработке, отладке и тестировании алгоритмов, а не на комплексном, трудоёмком и как следствие затратном по времени процессе сборки и формирования достаточно репрезентативных датасетов.

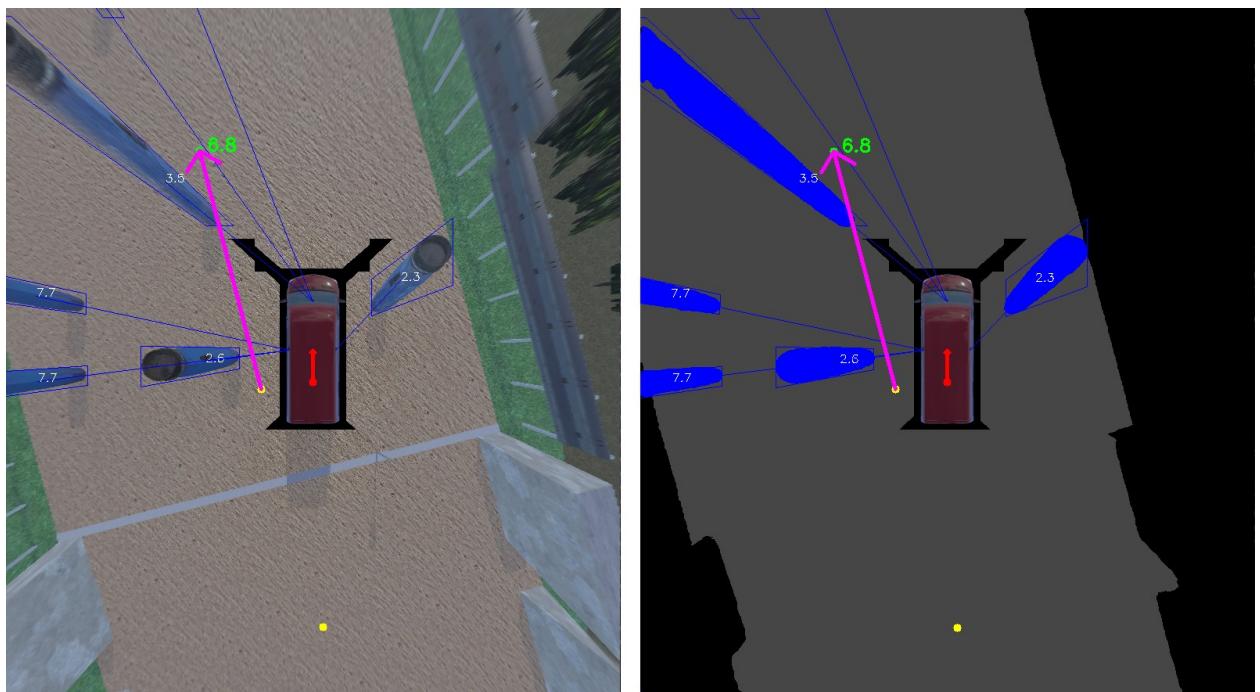


Рис. 44. Обычная (слева) и сегментированная (справа) локальная карта

Таким образом, данный эксперимент показал, что использование сегментированной локальной карты, с точки зрения точности и производительности моделей сегментации и детекции, применяемых для её формирования, пригодно для дальнейшей интеграции с форматом карты алгоритмов планирования маршрута движения эго-автомобиля.

3.4 Глобальная карта и планирование маршрута движения беспилотного наземного транспортного средства

3.4.1 Глобальная карта

3.4.1.1 Эксперимент №1

Данный эксперимент нацелен на то, чтобы с использованием данных с виртуального лидара и GPS с IMU, а также через взаимодействие с модулем «Navigation», построить и сохранить для дальнейшего использования точную глобальную карту сцены №3 в Webots (см. Рис. 34).

В левой части Рисунка 45 ниже представлена визуализация данных в формате PointCloud2 с виртуального лидара из Webots (правая часть Рисунка 45) в виде множества белых точек ([англ. point](#)), которые как бы образуют из себя облако ([англ. cloud](#)) – отсюда и название формата:

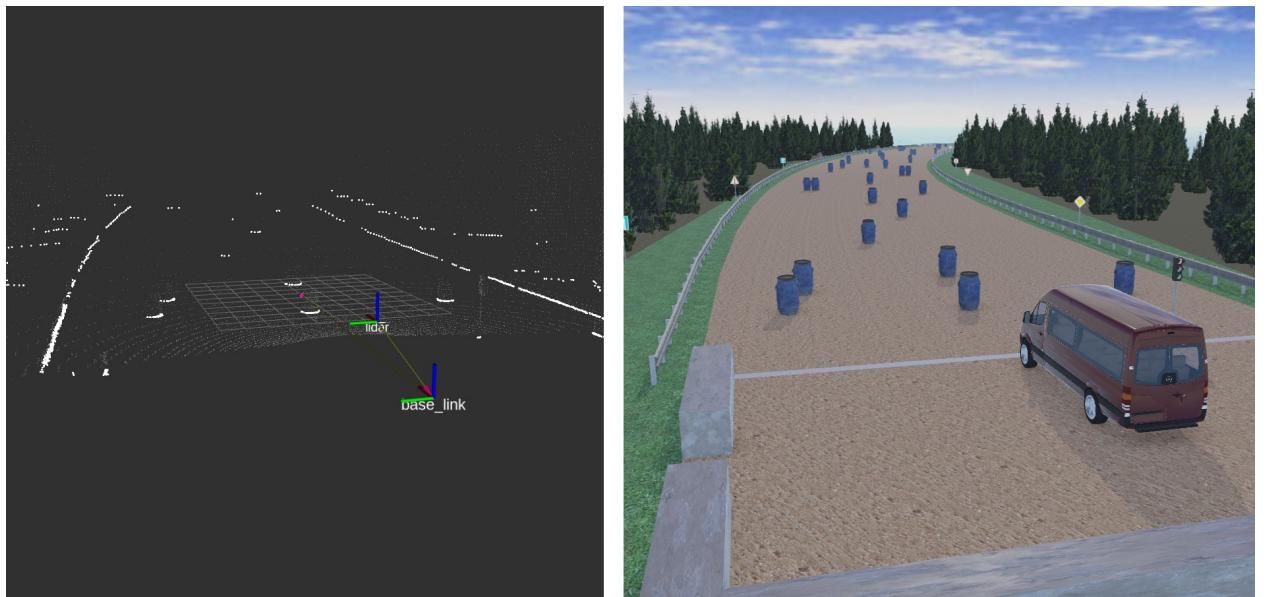


Рис. 45. Визуализация данных (PointCloud2 и LaserScan) с виртуального лидара из Webots

Более выделяющиеся белые точки – это результат работы пакета `pointcloud_to_laserscan`, который делает двухмерный срез в формате LaserScan одного из слоёв трёхмерного облака в формате PointCloud2 и настраивается через следующий набор параметров:

```

pointcloud_to_laserscan:
ros__parameters:
  min_height: 0.0 # Мин. и макс. высота в метрах, ниже и выше которой точки из PointCloud2 будут
  max_height: 0.75 # игнорироваться
  angle_min: -1.57 # Мин. и макс. углы сканирования в радианах, которые определяют левую и правую
  angle_max: 1.57 # границы поля обзора относительно лидара соответственно
  angle_increment: 0.00153 # Количество радиан на один лазерный луч – разрешение итогового LaserScan
  queue_size: 100 # Размер очереди входящих сообщений в формате PointCloud2
  scan_time: 0.01 # Время сканирования в секундах (обычно равно частоте обновления лидара)
  range_min: 0.75 # Мин. и макс. дистанция сканирования в метрах, ближе и дальше которой точки
  range_max: 100.0 # из PointCloud2 будут игнорироваться
  target_frame: 'base_link' # Фрейм, в который будет преобразован PointCloud2 перед его обработкой в LaserScan
  concurrency_level: 1 # Количество потоков, одновременно обрабатывающих облака точек с лидара
  transform_tolerance: 0.01 # Допуск по времени в секундах для ожидания трансформации между фреймами
  use_inf: true # Использовать значение inf или range_max + 1 для диапазонов сканирования без объектов

```

Листинг 8. Параметры пакета pointcloud_to_laserscan

Во встроенным в ROS 2 инструменте RViz для визуализации данных из топиков, оси с наименованиями фреймов «`base_link`» (то же, что и `base_footprint` на Рис. 47 ниже) и «`lidar`» (см. левую часть на Рис. 45) – это упрощённое представление элементов этого-автомобиля – его заднего моста и лидара соответственно.

RViz ([англ.](#) ROS Visualization) активно использовался при разработке, отладке и теперь уже при тестировании решения, поэтому все последующие изображения, по аналогии с Рисунком 45, будут представлять из себя связку RViz (слева) + Webots (справа).

На Рисунках 46-47 ниже представлен процесс построения глобальной карты полосы препятствий в симуляторе (см. Рис. 34) с использованием всего того же узла `ackermann_keyboard_teleop_node` для ручного управления этим автомобилем с клавиатуры, а также библиотеки SLAM Toolbox, описание работы которой, так же как и описание работы пакета `pointcloud_to_laserscan` и фреймворка Nav2, было представлено в разделе 2.5 «**Глобальная карта и планирование маршрута движения беспилотного наземного транспортного средства**» главы 2:

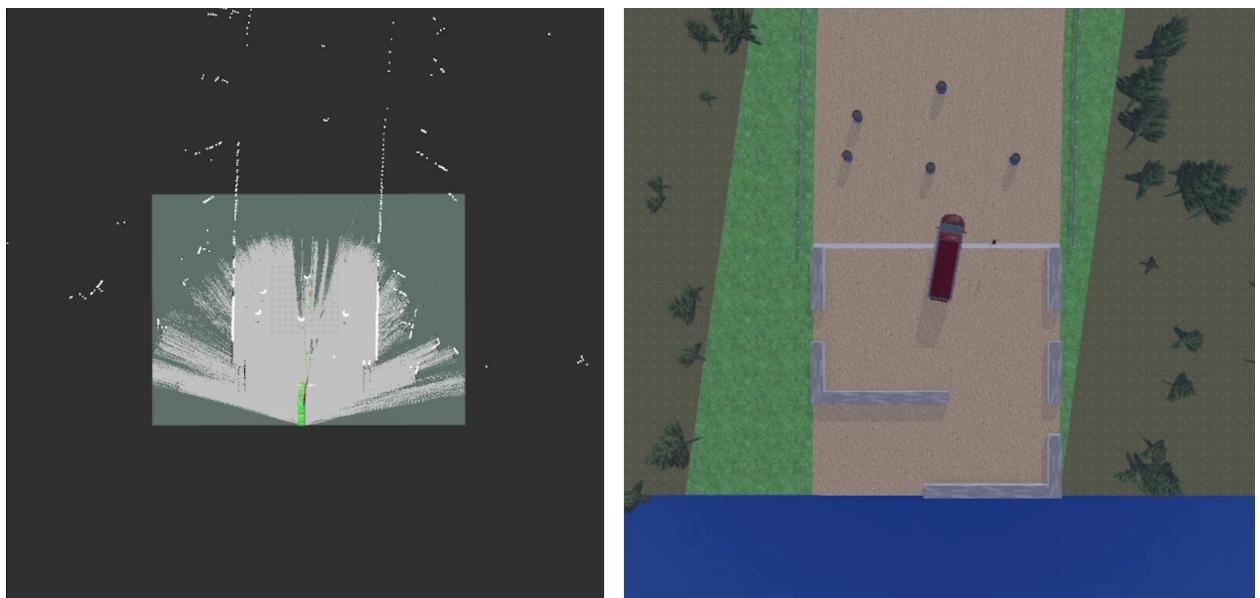


Рис. 46. Процесс построения глобальной карты с использованием SLAM Toolbox

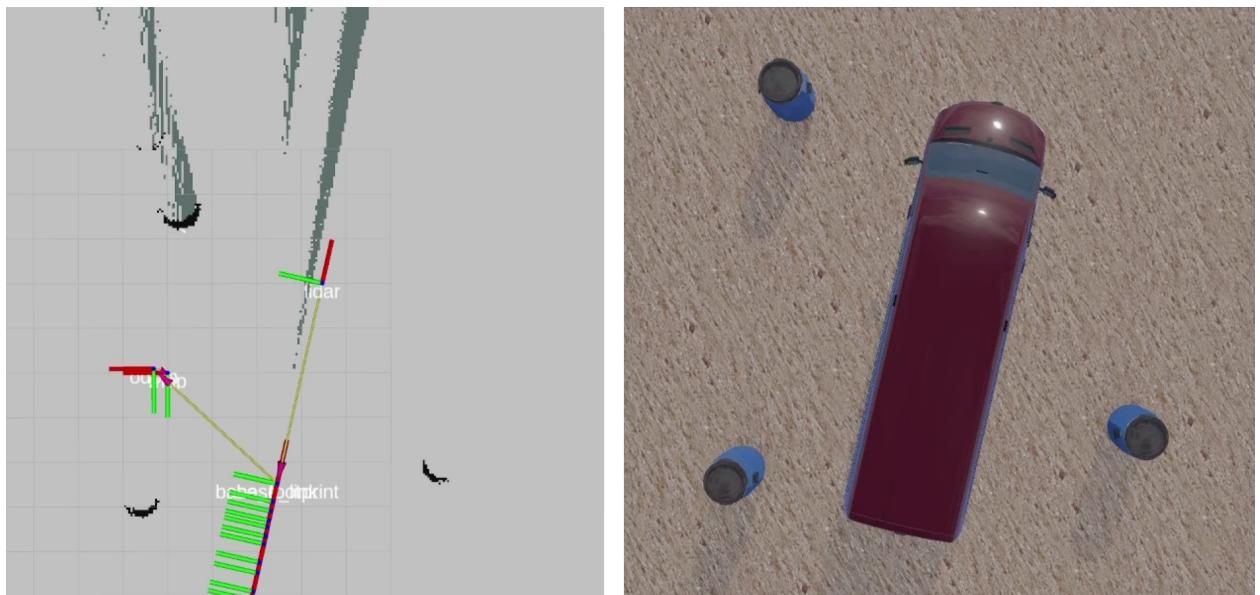


Рис. 47. Фреймы «map», «base_footprint» и «lidar», а также «одом» одометрии автомобиля

После того как автомобиль под управлением человека полностью проехал весь испытательный полигон сцены №3 в Webots от начала и до конца, а его глобальная карта в формате OccupancyGrid была построена, её необходимо сохранить. В RViz это делается следующим образом:

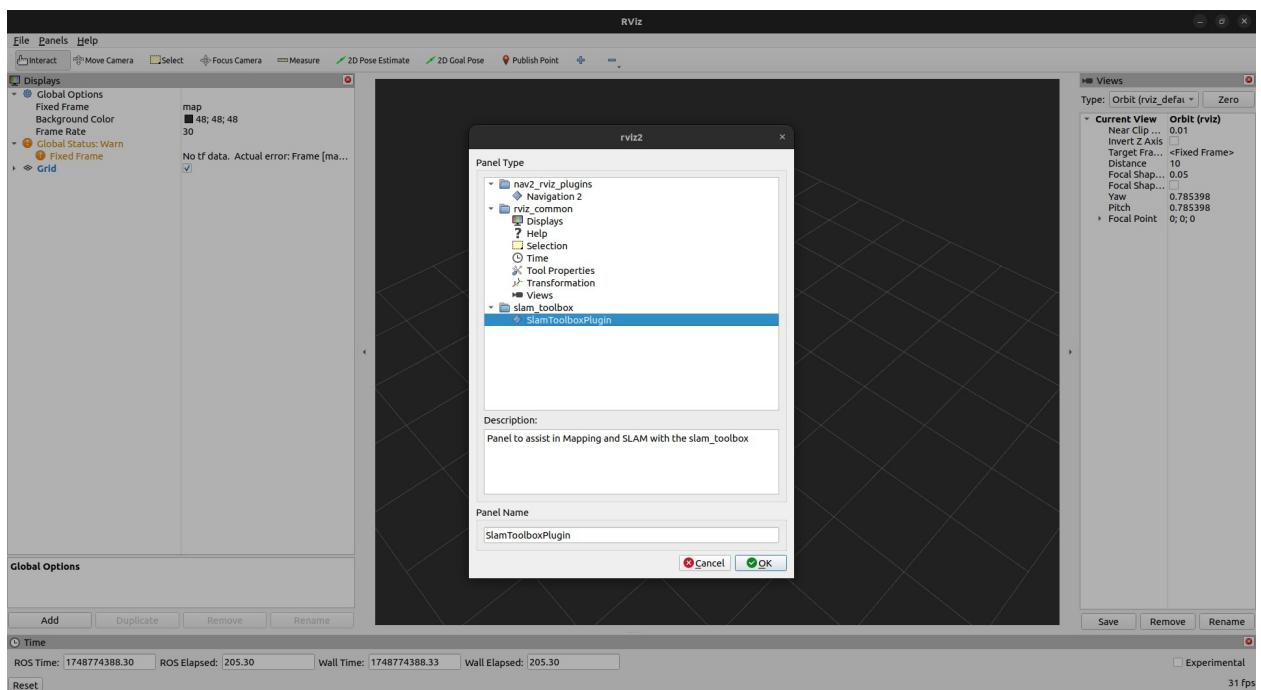


Рис. 48. В верхнем меню Panels → Add New Panel

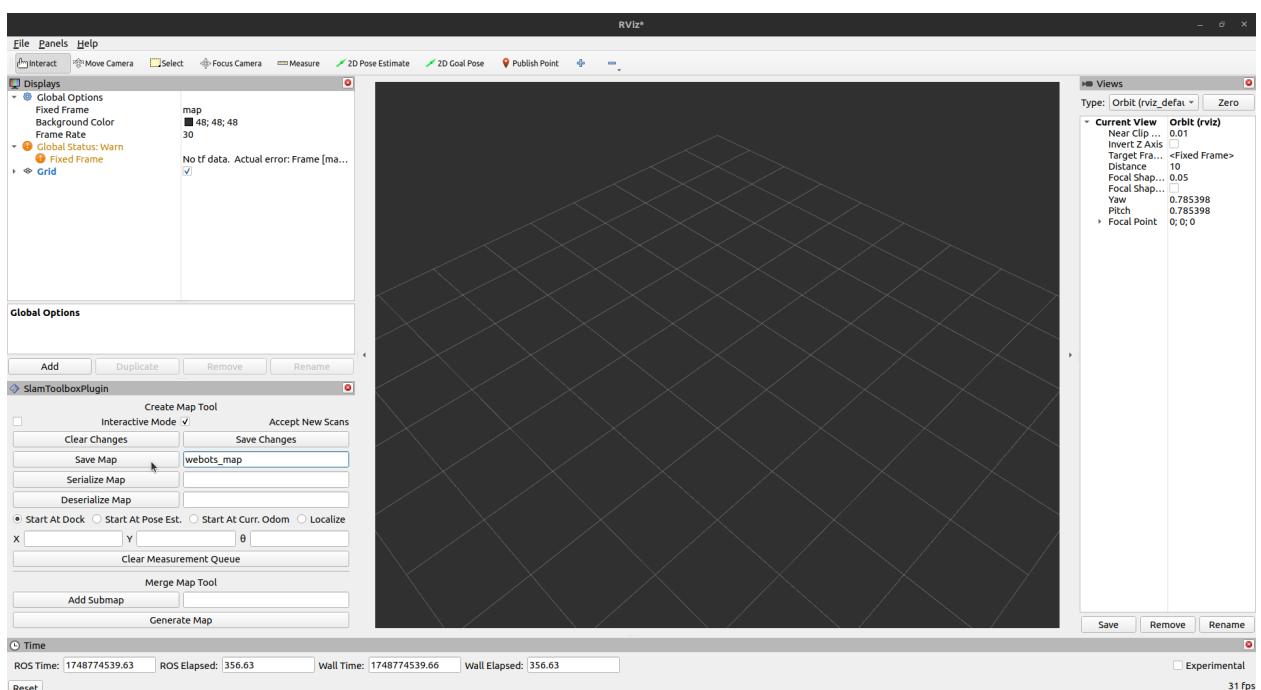


Рис. 49. Вводим первое название файла карты → Save Map

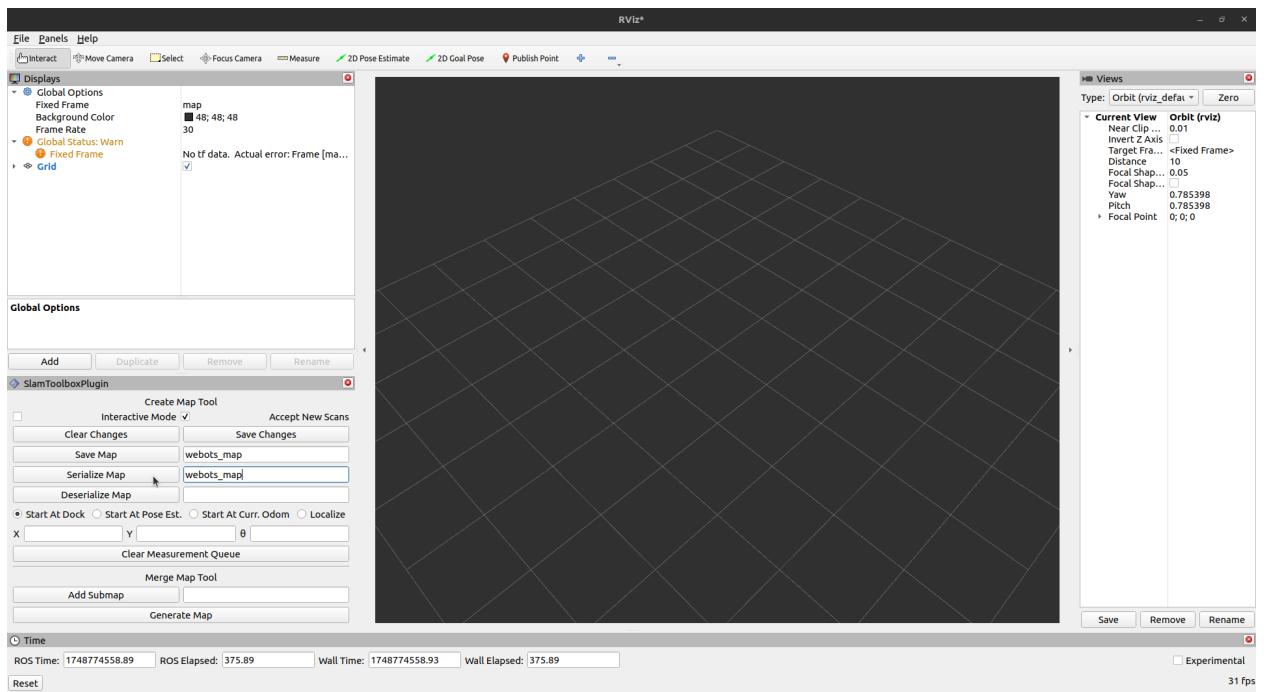


Рис. 50. Вводим второе название (может совпадать с первым) файла карты → Serialize Map

По умолчанию все файлы (.data, .pgm (см. Рис. 51), .posegraph и .yaml) сохраняются в ту директорию, в которой была запущена утилита.

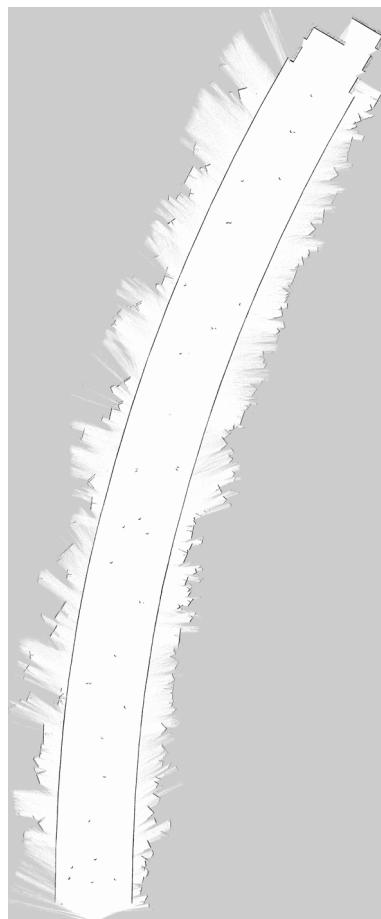


Рис. 51. Формат .pgm описания построенной глобальной карты

На Рисунках 46-47, множество осей, тянувшихся за виртуальным БНТС – это GPS-IMU-одометрия робота, которая должна примерно соответствовать его фактическому перемещению в мире Webots.

Всё дерево фреймов без применения статических трансформаций, каким оно должно быть для правильной работы не только SLAM Toolbox, но и Nav2, конкретно для используемой готовой 3D-модели автомобиля Mercedes-Benz Sprinter из Webots имеет следующую структуру типа предок-потомок:

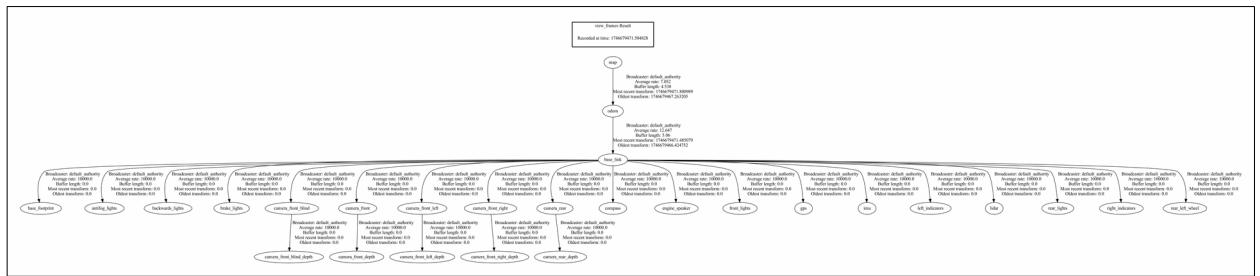


Рис. 52. Фреймы сверху вниз: «map» → «odom» → «base_link» → ... → «lidar» → ...

Таким образом, данный эксперимент показал, что с использованием данных с виртуального лидара и GPS с IMU, а также через взаимодействие с модулем «Navigation», можно без каких-либо проблем построить очень большую по площади (2150x5211 пикселей), но при этом и довольно точную глобальную карту не только сцены №3 в Webots (см. Рис. 34), но и любого другого виртуального мира, после чего удобно сохранить её сразу в нескольких форматах для дальнейшего использования.

3.4.2 Планирование маршрута движения БНТС

3.4.2.1 Эксперимент №1

Данный эксперимент нацелен на то, чтобы протестировать готовую реализацию алгоритма планирования глобального пути NavFn Planner из фреймворка Nav2 в условиях виртуального испытательного полигона сцены №3 в Webots (см. Рис. 34).

Предполагается, что в текущем и в двух оставшихся экспериментах БНТС будет использовать для планирования маршрута своего движения информацию с глобальной карты, построенной и сохранённой в формате OccupancyGrid ранее и загружаемой теперь через узел map_server, но уже в формате Costmap, в который он сам её и преобразует и который требуется планировщикам Nav2, полный перечень параметров которого, так же, как и самих планировщиков представлен в Листинге 12 **Приложения А** ниже.

Управляющие команды в **Экспериментах №1-3** будут отправляться в Webots компьютером, а не человеком.

На Рисунках 53-54 ниже представлен глобальный длинный путь в прямом и обратном направлениях соответственно, который NFP построил всего по двум точкам, отмеченным вручную с помощью инструмента под названием «2D Goal Pose» в утилите для визуализации RViz:

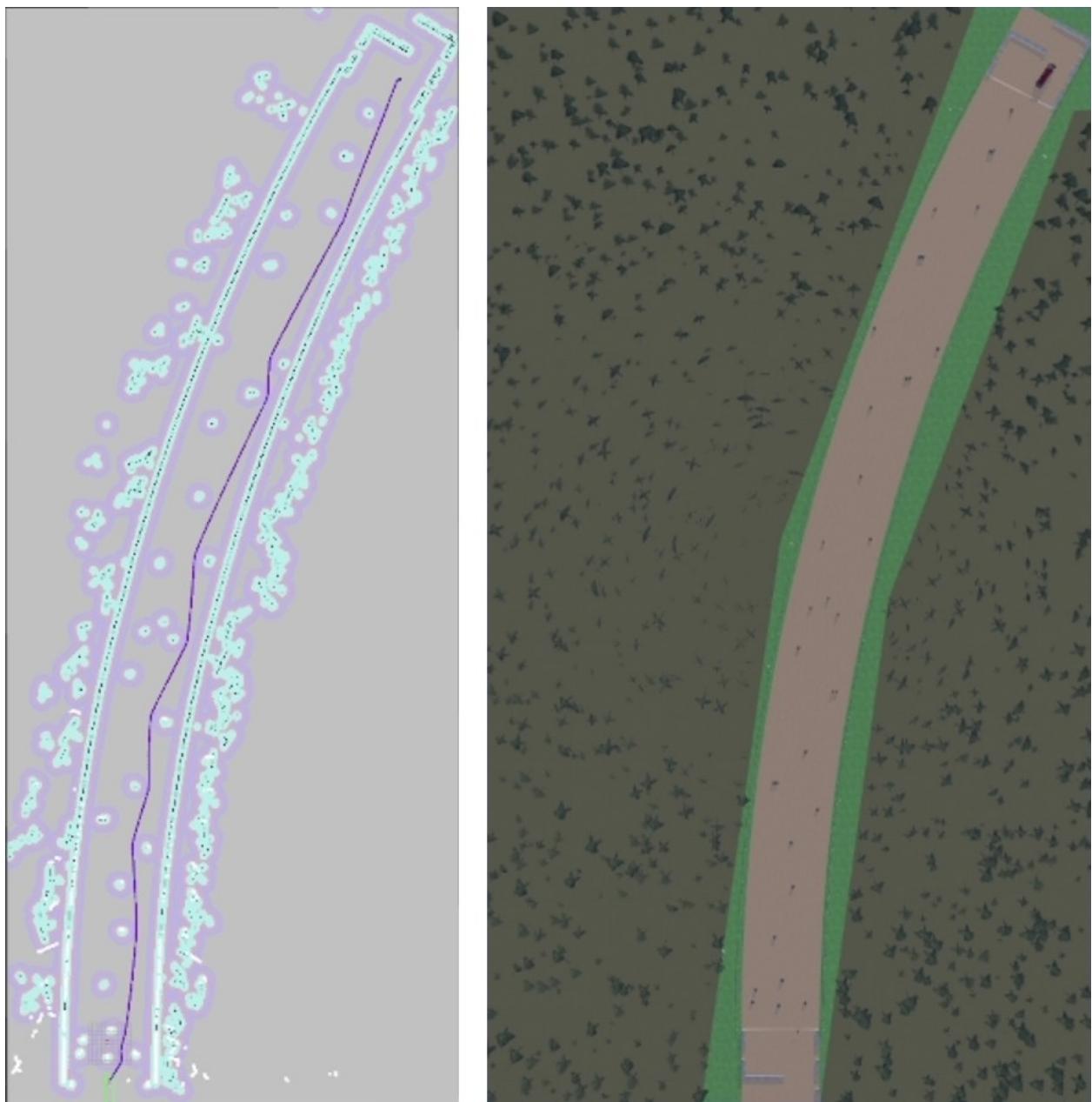


Рис. 53. Глобальный длинный прямой путь (фиолетовая линия слева), построенный NFP

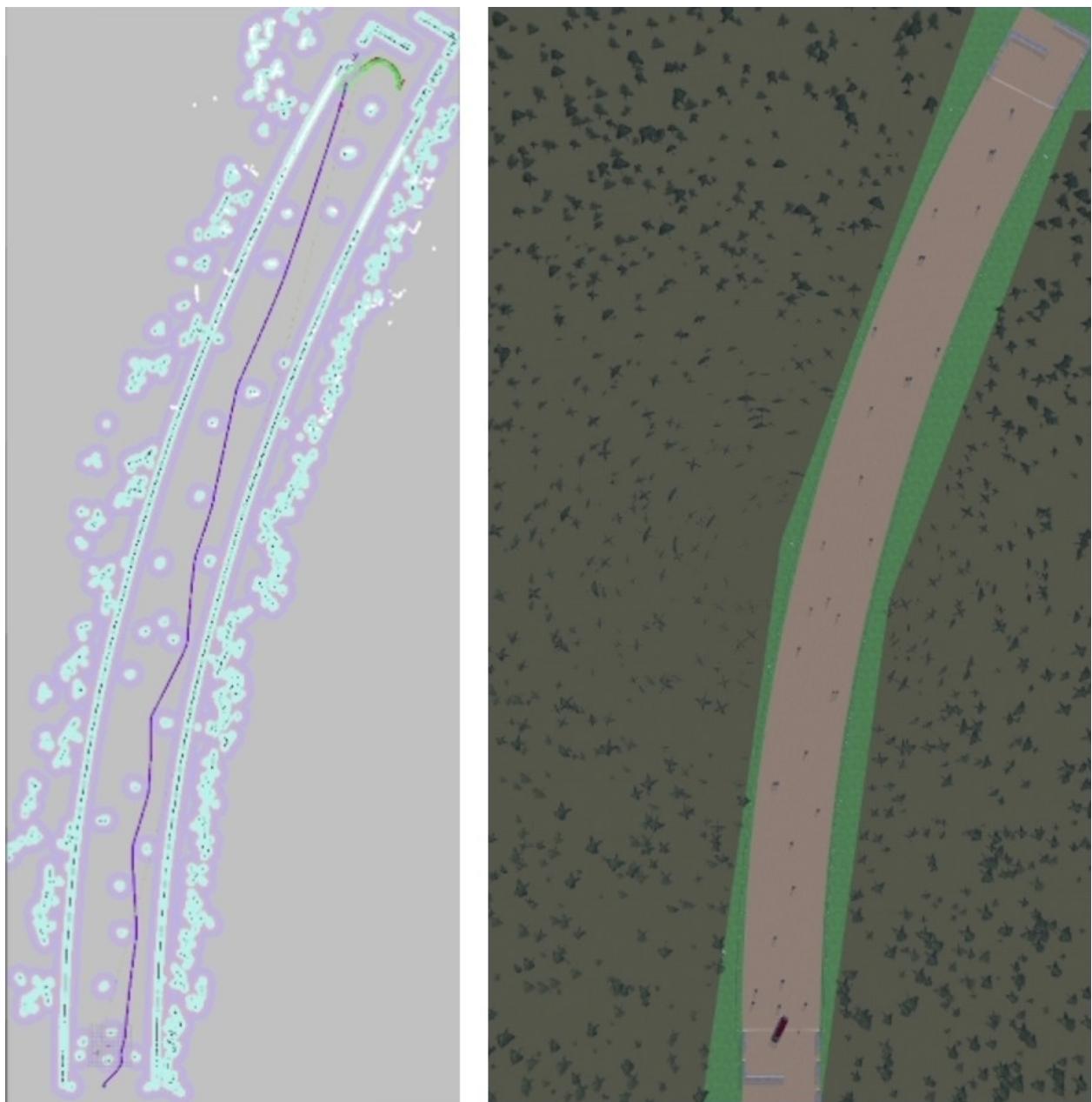


Рис. 54. Глобальный длинный обратный путь, построенный NFP

На Рисунке 55 ниже представлен глобальный короткий путь, который NFP построил по очередной точке из списка route (см. Лист. 2 выше), опубликованной в формате PoseStamped [47] узлом под названием «nav2_path_planning_node» с использованием таких встроенных классов, как ActionClient и NavigateToPose. Сам узел входит в состав модуля «Navigation» и имеет схожую с узлом gps_path_planning_node (см. раз. 2.4 «Сегментированная локальная карта») логику работы:

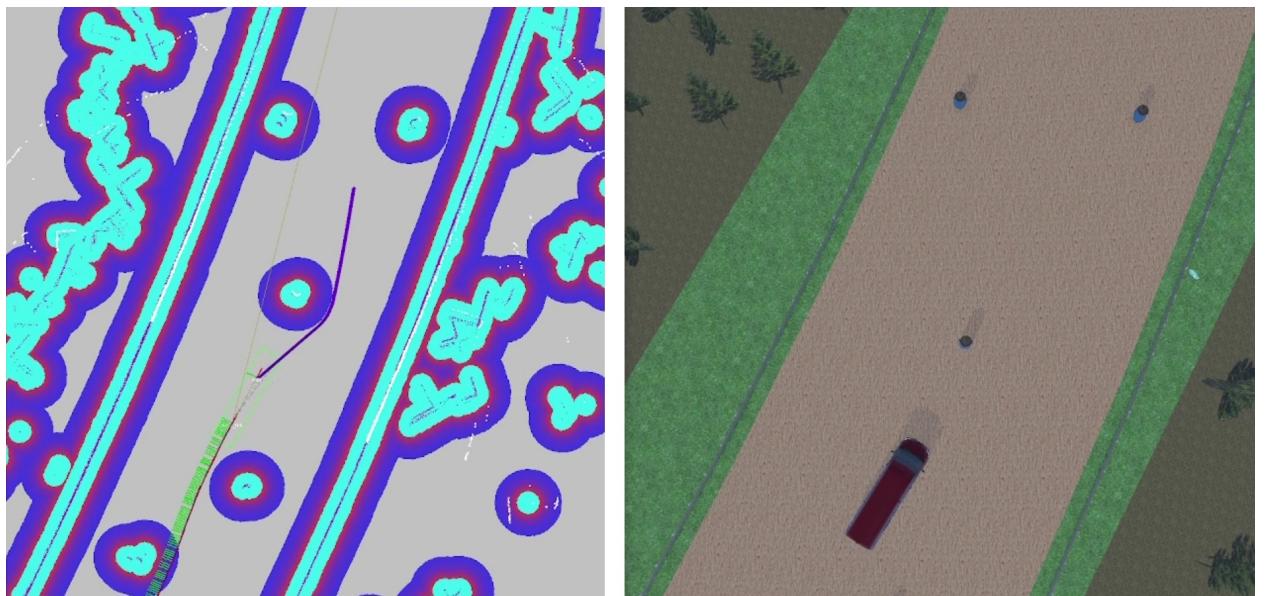


Рис. 55. Глобальный короткий путь, построенный NFP

Таким образом, данный эксперимент показал, что со значениями параметров алгоритма планирования глобального пути NavFn Planner по умолчанию, он хорошо справляется как с длинными, так и с короткими маршрутами, хотя это также зависит и от внешних факторов, таких как: качество исходной карты, размеры самого робота и как следствие всего этого – от качества итоговой карты стоимости ([англ. costmap](#)) и радиуса опасной зоны ([англ. inflation radius](#)) вокруг потенциальных препятствий, значение которого и определяет насколько близко к ним будет строиться путь.

3.4.2.2 Эксперимент №2

Данный эксперимент нацелен на то, чтобы протестировать готовую реализацию алгоритма следования локальному пути Regulated Pure Pursuit из Nav2 в условиях и предусловиях аналогичных Эксперименту №1.

Одним из ключевых параметров RPP для тестирования является `lookahead_dist`, который определяет, какое расстояние от робота в метрах будет просматривать данный алгоритм и следовательно, насколько рано, или наоборот поздно, он начнёт поворачивать в нужном направлении.

На Рисунках 56-58 ниже приведено сравнение результатов работы планировщика для трёх различных значений этого параметра в одном из наиболее труднопроходимых мест сцены №3:

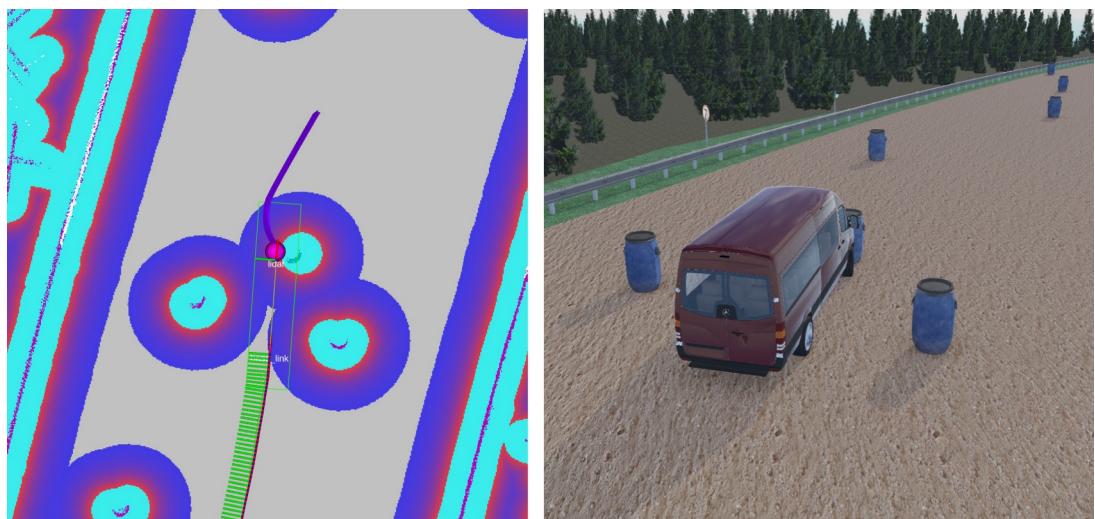


Рис. 56. RPP следует пути от NFP (слева) при $\text{lookahead_dist} = 5 \rightarrow$ контакт с препятствием

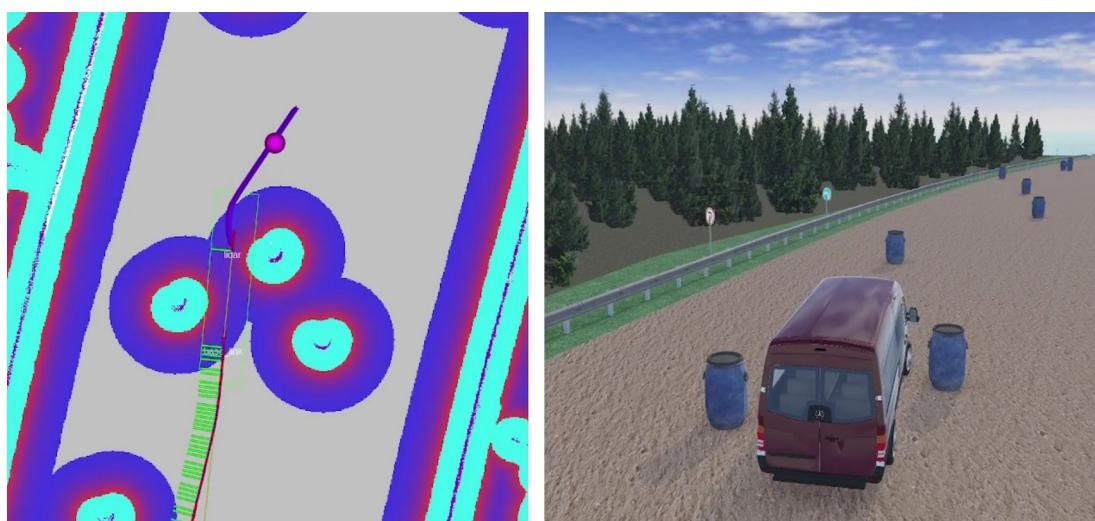


Рис. 57. RPP следует пути от NFP при $\text{lookahead_dist} = 10 \rightarrow$ нет контакта с препятствием

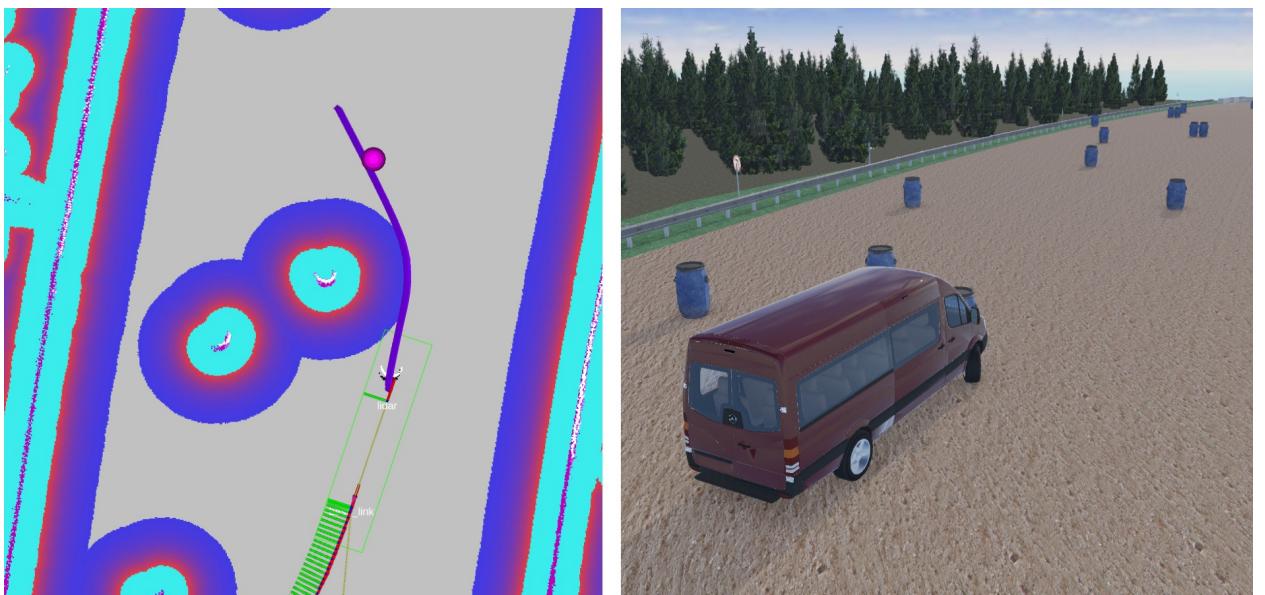


Рис. 58. RPP следует пути от NFP при lookahead_dist = 15 → контакт с препятствием

Таким образом, данный эксперимент показал, что связка одного глобального и одного локального планировщиков NavFn Planner + Regulated Pure Pursuit, с грамотно подобранными параметрами (см. Лист. 12), неплохо показала себя, задев всего одно препятствие из 33-х на протяжении всего маршрута движения эго-автомобиля через полосу препятствий в симуляторе (см. Рис. 34).

3.4.2.3 Эксперимент №3

Данный эксперимент нацелен на то, чтобы протестировать готовую реализацию алгоритма локального контроллера Model Predictive Path Integral при всё тех же вводных, что и ранее.

Одними из ключевых параметров MPPI для тестирования являются batch_size, который определяет количество случайных траекторий-кандидатов, прогоняемых через динамическую модель робота за одну итерацию алгоритма и time_steps, который определяет количество точек в этих траекториях.

Оба параметра влияют на так называемый горизонт планирования, а увеличение их значений приводит к значительному приросту вычислительной нагрузки на всю систему, что, впрочем, закономерно.

На Рисунках 59-65 ниже приведено сравнение результатов работы планировщика для трёх различных пар значений этих параметров в одном из наиболее труднопроходимых мест сцены №3:

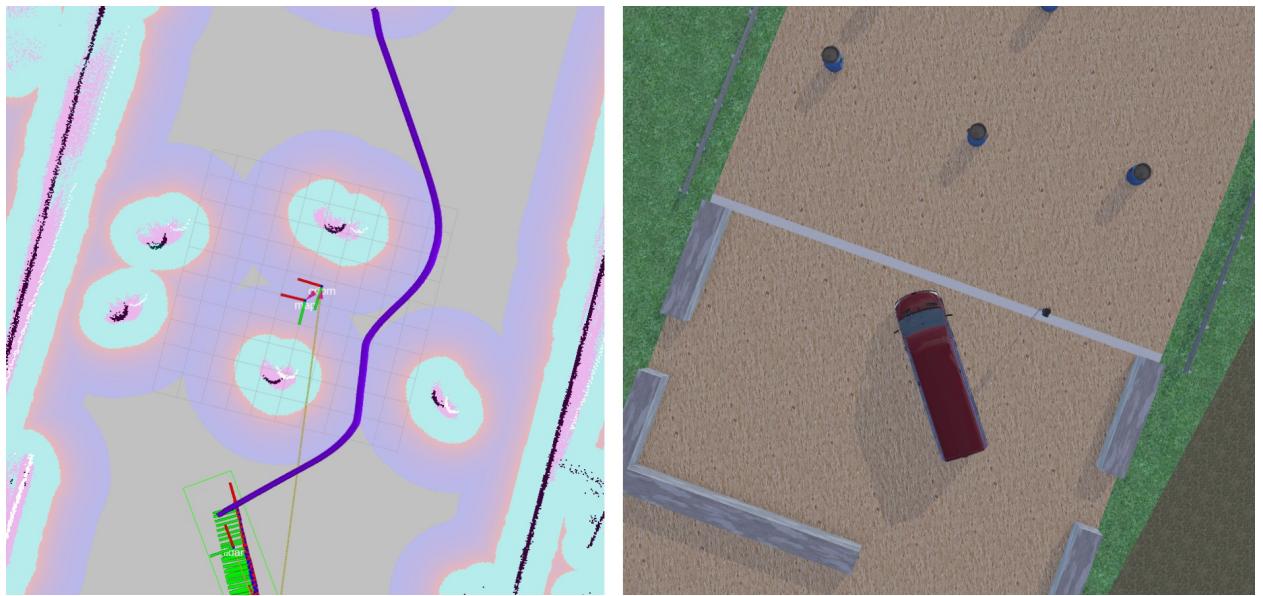


Рис. 59. MPPI следует пути от NFP (слева) при $\text{batch_size} = 500$ и $\text{time_steps} = 28$

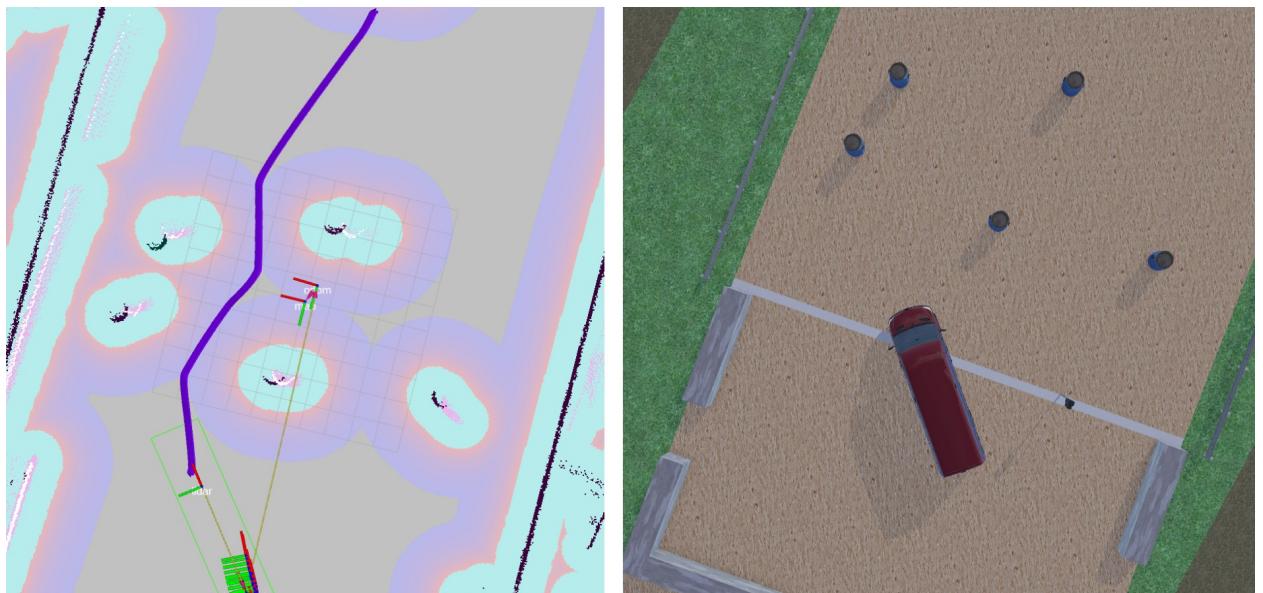


Рис. 60. MPPI следует пути от NFP при $\text{batch_size} = 500$ и $\text{time_steps} = 28$

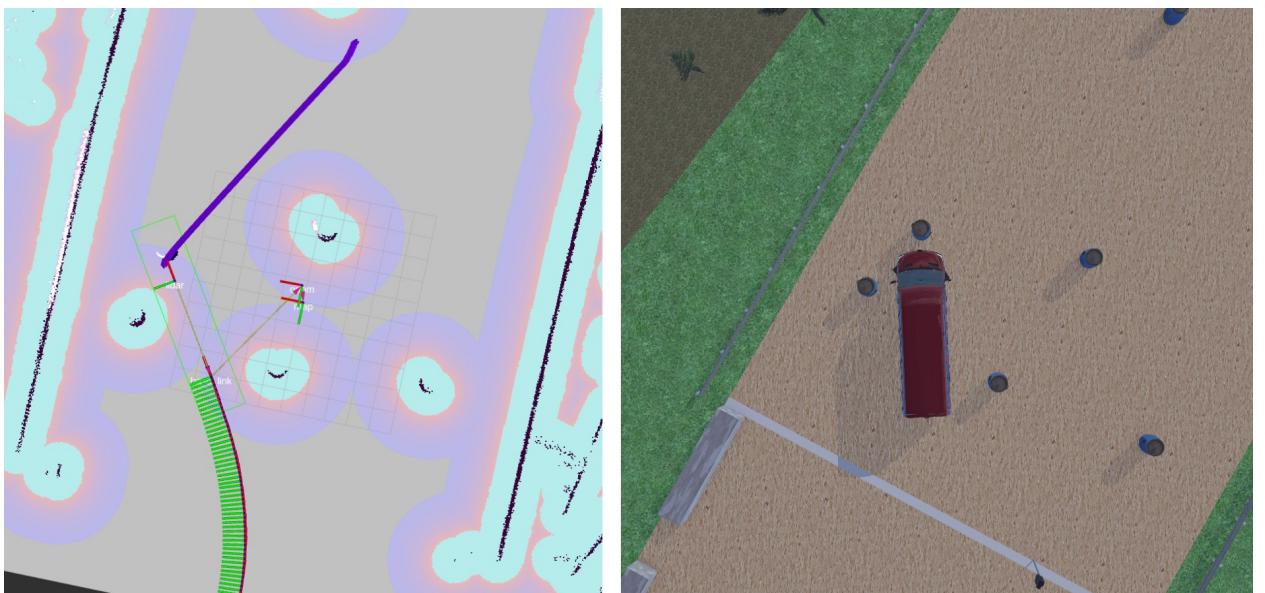


Рис. 61. batch_size = 500 и time_steps = 28 → БНТС не смог выехать со старовой локации

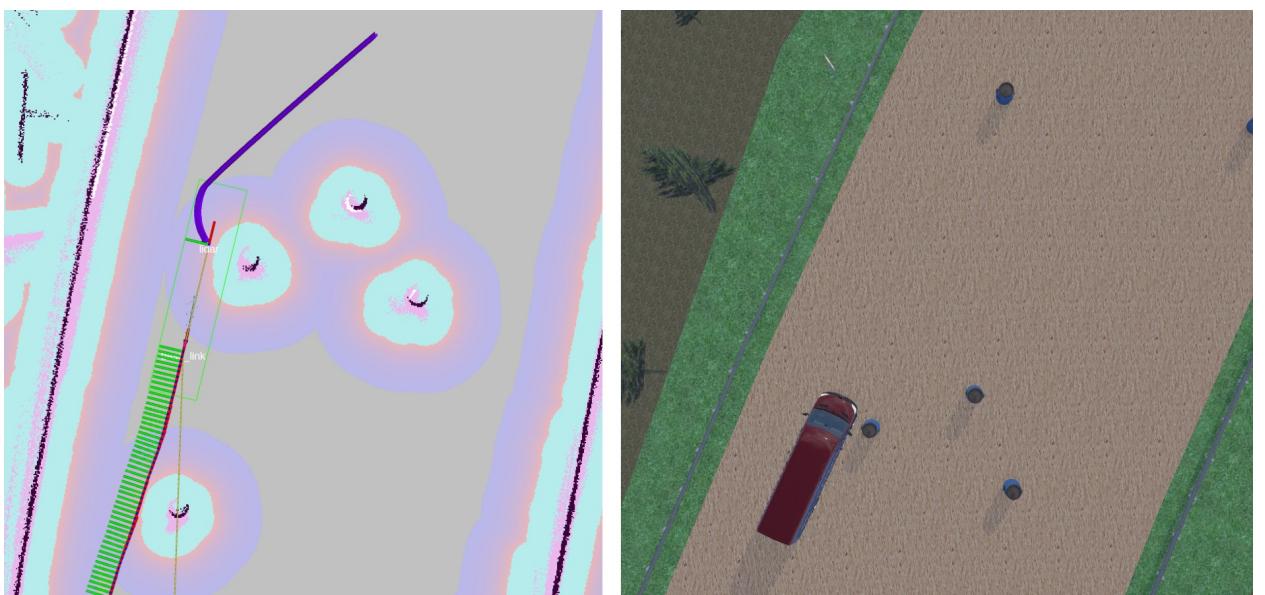


Рис. 62. batch_size = 1000 и time_steps = 56 → нет контакта с препятствием

Несмотря на то, что при значениях по умолчанию параметров batch_size и time_steps в 1000 и 56 соответственно у этого-автомобиля всё-таки не произошло контакта с препятствием (см. Рис. 62), на других участках своего маршрута он приближался к бочкам очень близко перед тем как начинать их объезжать, но могло быть уже слишком поздно для совершения настолько резкого манёвра:

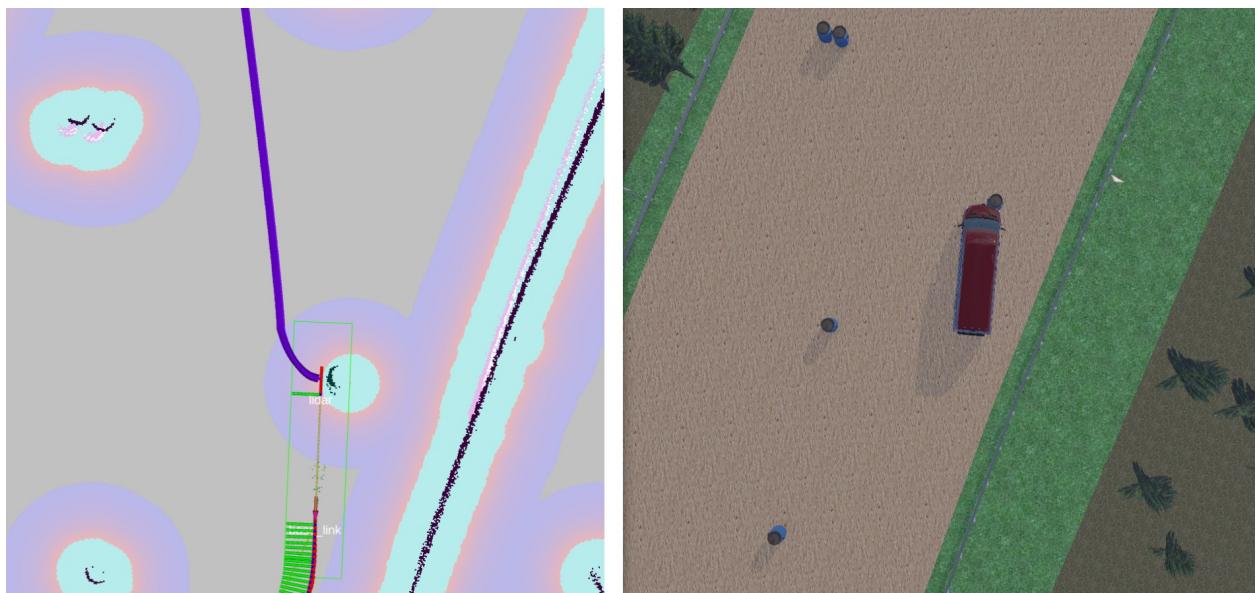


Рис. 63. batch_size = 1000 и time_steps = 56 → контакт с препятствием

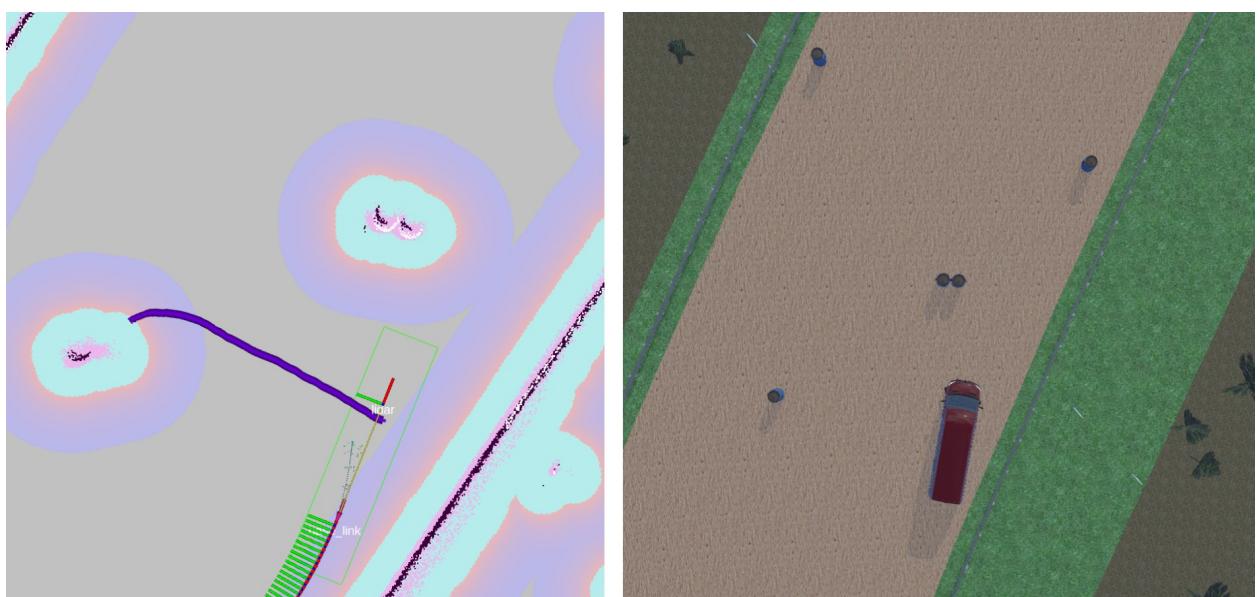


Рис. 64. batch_size = 1000 и time_steps = 56 → БНТС начал поздно поворачивать

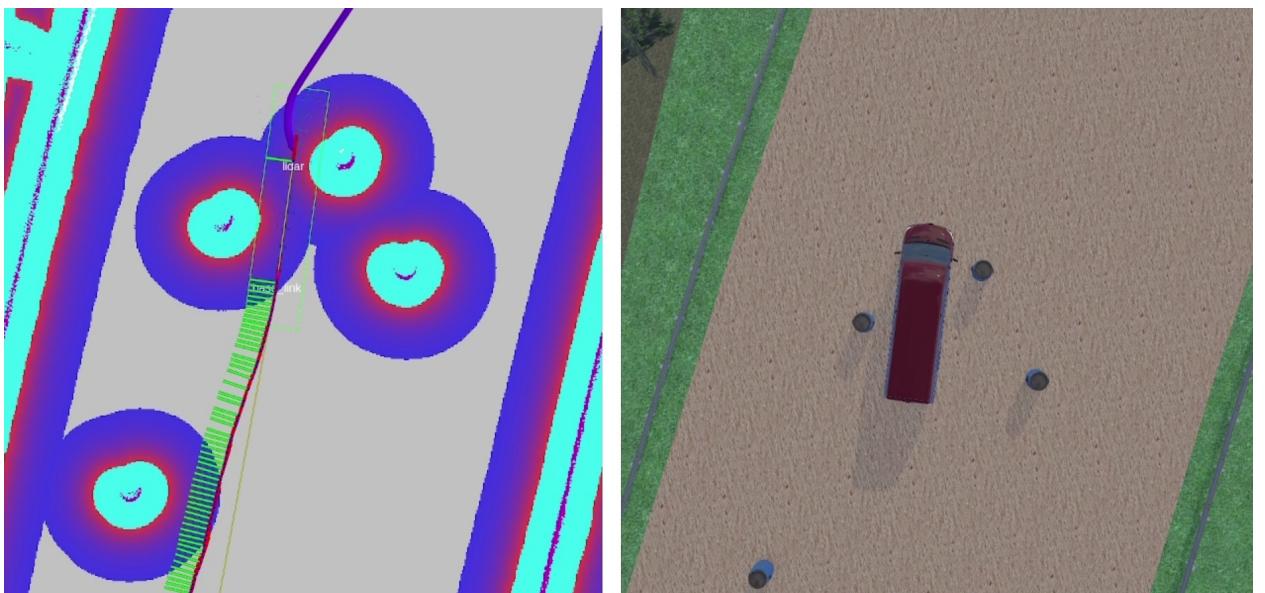


Рис. 65. $\text{batch_size} = 2000$ и $\text{time_steps} = 125 \rightarrow$ нет контакта с препятствием

Таким образом, данный эксперимент показал, что связка одного глобального планировщика и одного локального контроллера NavFn Planner + Model Predictive Path Integral, с грамотно подобранными параметрами (см. Лист. 12), неплохо показала себя, так же задев всего одно препятствие из 33-х., как этого удалось достичь и при использовании связки NavFn Planner + Regulated Pure Pursuit в предыдущем **Эксперименте №2**.

Несмотря на это, была отмечена значительно возросшая вычислительная нагрузка на систему, которая (система) использовалась и оставалась неизменной на протяжении всего этапа тестирования решения и во всех проводимых экспериментах.

В связи с этим, а также по причине, указанной в подразделе 2.5.2 соответствующего планировщика в предыдущей главе 2, MPPI следовал локальному пути не так стабильно, как это делал RPP, а также мог просто без видимой на то причины в принципе переставать ему следовать, что приводило к ненужным остановкам и «раздумьям» беспилотного наземного транспортного средства прямо посреди полигона в Webots.

Заключение

По итогу выполнения данной работы, было разработано решение, которое позволяет на основе системы двухмерного кругового обзора строить сегментированную локальную карту и планировать по ней маршрут движения беспилотного наземного транспортного средства в виртуальной среде симуляции [48].

В процессе были проанализированы существующие подходы к восприятию БНТС окружающего пространства и их применение при планировании маршрута движения; подготовлена виртуальная среда симуляции с возможностью калибровки используемых в ней стереокамер; на основе данных с них была разработана система двухмерного кругового обзора, а затем построена сегментированная локальная карта с применением инструментов машинного и глубокого обучений.

Объединение предложенных готовых алгоритмов планирования и следования маршруту движения с форматом построенной локальной карты – это та задача, которую только предстоит выполнить, хотя большая часть наработок по ней уже в том или ином виде присутствует в данном, а также в сторонних решениях собственной разработки.

Дальнейшие возможные исследования могут быть направлены на интеграцию предложенного подхода с трёхмерными системами кругового обзора и разработку гибридных моделей [49], сочетающих в себе различные источники данных для повышения качества картирования и навигации беспилотных наземных, а может быть и не только, транспортных средств.

Список используемой литературы

1. Tadviser – портал выбора технологий и поставщиков, «Беспилотные автомобили в России». – 2025. – [Электронный ресурс]. – Режим доступа: https://www.tadviser.ru/index.php/Статья:Беспилотные_автомобили_в_России (дата обращения: 25.05.2025);
2. Tadviser – портал выбора технологий и поставщиков, «Беспилотные автомобили (мировой рынок)». – 2024. – [Электронный ресурс]. – Режим доступа: [https://www.tadviser.ru/index.php/Статья:Беспилотные_автомобили_\(мировой_рынок\)](https://www.tadviser.ru/index.php/Статья:Беспилотные_автомобили_(мировой_рынок)) (дата обращения: 25.05.2025);
3. D. J. Yeong, G. Velasco-Hernandez, J. Barry, J. Walsh, «Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review,» *Sensors*. – 2021. – Vol. 21, No. 6. – Article 2140. – [Электронный ресурс]. – Режим доступа: <https://www.mdpi.com/1424-8220/21/6/2140> (дата обращения: 12.05.2025);
4. ApolloAuto/apollo, «An open autonomous driving platform» – [Электронный ресурс]. – Режим доступа: <https://github.com/ApolloAuto/apollo> (дата обращения: 25.05.2025);
5. autowarefoundation/autoware, «Autoware - the world's leading open-source software project for autonomous driving» – [Электронный ресурс]. – Режим доступа: <https://github.com/autowarefoundation/autoware> (дата обращения: 25.05.2025);
6. carla-simulator/carla, «Open-source simulator for autonomous driving research.» – [Электронный ресурс]. – Режим доступа: <https://github.com/carla-simulator/carla> (дата обращения: 25.05.2025);
7. What Is Isaac Sim? – [Электронный ресурс]. – Режим доступа: <https://docs.isaacsim.omniverse.nvidia.com/latest/index.html> (дата обращения: 25.05.2025);

8. Gazebo – [Электронный ресурс]. – Режим доступа: <https://gazebosim.org/home> (дата обращения: 25.05.2025);
9. lgsvl/simulator, «A ROS/ROS2 Multi-robot Simulator for Autonomous Vehicles» – [Электронный ресурс]. – Режим доступа: <https://github.com/lgsvl/simulator> (дата обращения: 25.05.2025);
10. Welcome to Python.org – [Электронный ресурс]. – Режим доступа: <https://www.python.org/> (дата обращения: 25.05.2025);
11. OpenCV – Open Computer Vision Library – [Электронный ресурс]. – Режим доступа: <https://opencv.org/> (дата обращения: 13.05.2025);
12. J. Redmon, S. Divvala, R. Girshick, A. Farhadi, «You Only Look Once: Unified, Real-Time Object Detection,» – 2015. – [Электронный ресурс]. – Режим доступа: <https://arxiv.org/abs/1506.02640> (дата обращения: 14.05.2025);
13. E. Zhang, «Fast Semantic Segmentation» – [Электронный ресурс]. – Режим доступа: <https://github.com/ekzhang/fastseg> (дата обращения: 14.05.2025);
14. ROS: Home – [Электронный ресурс]. – Режим доступа: <https://www.ros.org/> (дата обращения: 12.05.2025);
15. Cyberbotics: Robotics simulation with Webots – [Электронный ресурс]. – Режим доступа: <https://cyberbotics.com/> (дата обращения: 12.05.2025);
16. Macenski et al., «SLAM Toolbox: SLAM for the dynamic world,» Journal of Open Source Software. – 2021. – 6(61). – 2783. – [Электронный ресурс]. – Режим доступа: <https://doi.org/10.21105/joss.02783> (дата обращения: 15.05.2025);
17. Nav2 Navigation System – [Электронный ресурс]. – Режим доступа: <https://nav2.org/> (дата обращения: 15.05.2025);

18. D. C. Brown, «Decentering distortion of lenses,» Photogrammetric Engineering. – 1966. – Vol. 32, No. 3. – P. 444–462. – [Электронный ресурс]. – Режим доступа: https://web.archive.org/web/20180312205006/https://www.asprs.org/wp-content/uploads/pers/1966journal/may/1966_may_444-462.pdf (дата обращения: 13.05.2025);
19. Cambridge in Colour – Photography Tutorials & Learning Community, «Коррекция искажений, вносимых объективом» – [Электронный ресурс]. – Режим доступа: <https://www.cambridgeincolour.com/ru/tutorials-ru/lens-corrections.htm> (дата обращения: 13.05.2025);
20. H. Yan, «Surround View System Introduction» – [Электронный ресурс]. – Режим доступа: <https://github.com/hynpu/surround-view-system-introduction> (дата обращения: 14.05.2025);
21. Robot - Webots documentation – [Электронный ресурс]. – Режим доступа: <https://cyberbotics.com/doc/reference/robot> (дата обращения: 22.05.2025);
22. msg - ROS Wiki – [Электронный ресурс]. – Режим доступа: <https://wiki.ros.org/msg> (дата обращения: 22.05.2025);
23. ackermann_msgs/AckermannDrive Documentation – [Электронный ресурс]. – Режим доступа: https://docs.ros.org/en/jade/api/ackermann_msgs/html/msg/AckermannDrive.html (дата обращения: 22.05.2025);
24. sensor_msgs/MagneticField Documentation – [Электронный ресурс]. – Режим доступа: https://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/MagneticField.html (дата обращения: 22.05.2025);
25. std_msgs/Header Documentation – [Электронный ресурс]. – Режим доступа: https://docs.ros.org/en/noetic/api/std_msgs/html/msg/Header.html (дата обращения: 22.05.2025);

26. geometry_msgs/Quaternion Documentation – [Электронный ресурс]. – Режим доступа: http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Quaternion.html (дата обращения: 22.05.2025);
27. sensor_msgs/NavSatFix Documentation – [Электронный ресурс]. – Режим доступа: http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/NavSatFix.html (дата обращения: 22.05.2025);
28. CB64S1 Wide FOV LiDAR || Leishen Intelligent System – [Электронный ресурс]. – Режим доступа: <https://www.lslidar.com/product/ch64w-wide-fov-lidar/> (дата обращения: 22.05.2025);
29. geometry_msgs/TransformStamped Documentation – [Электронный ресурс]. – Режим доступа: http://docs.ros.org/en/jade/api/geometry_msgs/html/msg/TransformStamped.html (дата обращения: 23.05.2025);
30. nav_msgs/Odometry Documentation – [Электронный ресурс]. – Режим доступа: http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html (дата обращения: 23.05.2025);
31. geometry_msgs/PoseWithCovarianceStamped Documentation – [Электронный ресурс]. – Режим доступа: http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/PoseWithCovarianceStamped.html (дата обращения: 23.05.2025);
32. geometry_msgs/Twist Documentation – [Электронный ресурс]. – Режим доступа: http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Twist.html (дата обращения: 23.05.2025);
33. H. Bai, «ICP Algorithm: Theory, Practice And Its SLAM-oriented Taxonomy,» – 2022. – [Электронный ресурс]. – Режим доступа: <https://arxiv.org/abs/2206.06435> (дата обращения: 25.05.2025);
34. ceres-solver/ceres-solver, «A large scale non-linear optimization library» – [Электронный ресурс]. – Режим доступа: <https://github.com/ceres-solver/ceres-solver> (дата обращения: 25.05.2025);

35. nav_msgs/OccupancyGrid Documentation – [Электронный ресурс]. – Режим доступа: http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/OccupancyGrid.html (дата обращения: 23.05.2025);
36. NavFn Planner – Nav2 1.0.0 documentation – [Электронный ресурс]. – Режим доступа: <https://docs.nav2.org/configuration/packages/configuring-navfn.html> (дата обращения: 24.05.2025);
37. S. Macenski, S. Singh, F. Martin, J. Gines, «Regulated Pure Pursuit for Robot Path Tracking,» – 2023. – [Электронный ресурс]. – Режим доступа: <https://arxiv.org/abs/2305.20026> (дата обращения: 15.05.2025);
38. G. Williams, P. Drews, B. Goldfain, J. M. Rehg, E. A. Theodorou, «Aggressive driving with model predictive path integral control,» IEEE International Conference on Robotics and Automation (ICRA). – 2016. – [Электронный ресурс]. – Режим доступа: <https://ieeexplore.ieee.org/document/7487277> (дата обращения: 15.05.2025);
39. teb_local_planner - ROS Wiki – [Электронный ресурс]. – Режим доступа: https://wiki.ros.org/teb_local_planner (дата обращения: 15.05.2025);
40. C. Rösmann, A. Makarow, T. Bertram, «Online Motion Planning based on Nonlinear Model Predictive Control with Non-Euclidean Rotation Groups,» – 2020. – [Электронный ресурс]. – Режим доступа: <https://arxiv.org/abs/2006.03534> (дата обращения: 15.05.2025);
41. Z. Zhengyou, «A Flexible New Technique for Camera Calibration» – [Электронный ресурс]. – Режим доступа: <https://www.microsoft.com/en-us/research/publication/a-flexible-new-technique-for-camera-calibration/> (дата обращения: 13.05.2025);
42. ZED 2 - AI Stereo Camera – [Электронный ресурс]. – Режим доступа: <https://www.stereolabs.com/en-fi/products/zed-2> (дата обращения: 14.05.2025);

43. T. Merkulova, B. Jayakumar, «Evaluation framework for Image Segmentation Algorithms,» – 2025. – [Электронный ресурс]. – Режим доступа: <https://arxiv.org/abs/2504.04435> (дата обращения: 14.05.2025);
44. CVAT: Leading Image & Video Data Annotation Platform – [Электронный ресурс]. – Режим доступа: <https://www.cvat.ai/> (дата обращения: 14.05.2025);
45. Kaggle: Your Machine Learning and Data Science Community – [Электронный ресурс]. – Режим доступа: <https://www.kaggle.com/> (дата обращения: 14.05.2025);
46. K. Robison, «OpenAI cofounder Ilya Sutskever says the way AI is built is about to change,» – 2024. – [Электронный ресурс]. – Режим доступа: <https://www.theverge.com/2024/12/13/24320811/what-ilya-sutskever-sees-openai-model-data-training> (дата обращения: 14.05.2025);
47. geometry_msgs/PoseStamped Documentation – [Электронный ресурс]. – Режим доступа: https://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/PoseStamped.html (дата обращения: 01.06.2025);
48. A. Molchanov, «Surround View SegBEV» – [Электронный ресурс]. – Режим доступа: <https://github.com/ghub-ayrtom/surround-view-segbev> (дата обращения: 16.05.2025);
49. Z. Liu, H. Tang, A. Amini, X. Yang, H. Mao, D. Rus, S. Han, «BEVFusion: Multi-Task Multi-Sensor Fusion with Unified Bird's-Eye View Representation,» – 2022. – [Электронный ресурс]. – Режим доступа: <https://arxiv.org/abs/2205.13542> (дата обращения: 15.05.2025).

Приложение А. Листинги программного кода

Листинг 9. calibration_flags

```
calibration_flags = (
    # cv2.CALIB_USE_INTRINSIC_GUESS + # Использовать заданные внутренние параметры ( $f_x$ ,  $f_y$ ,  $c_x$ ,  $c_y$ ), а также коэффициенты радиальной и тангенциальной дисторсий в качестве начальных и оптимизировать их в процессе калибровки
    # cv2.CALIB_USE_EXTRINSIC_GUESS + # Использовать заданные внешние параметры (rotation_vectors и translation_vectors) в качестве начальных и оптимизировать их в процессе калибровки
    cv2.CALIB_FIX_PRINCIPAL_POINT + # Не изменять оптический центр линзы во время глобальной оптимизации
    cv2.CALIB_FIX_ASPECT_RATIO + # Зафиксировать соотношение сторон фокусных расстояний по осям X и Y (пиксели видеокамеры квадратные)
    # cv2.CALIB_ZERO_TANGENT_DIST + # Обнулить коэффициенты  $r_1$  и  $r_2$  тангенциальной дисторсии и не пытаться подобрать их во время оптимизации
    # cv2.CALIB_FIX_FOCAL_LENGTH + # Зафиксировать заранее известное значение фокусного расстояния (focal length). Требуется установка флага cv2.CALIB_USE_INTRINSIC_GUESS
    cv2.CALIB_FIX_K3 # Зафиксировать соответствующий коэффициент радиальной дисторсии и не пытаться подобрать его во время оптимизации
    # cv2.CALIB_RATIONAL_MODEL + # Использовать рациональную модель калибровки, которая учитывает коэффициенты  $k_4$ ,  $k_5$  и  $k_6$ , а также возвращает 8 или более коэффициентов радиальной дисторсии
    # cv2.CALIB_THIN_PRISM_MODEL + # Использовать модель калибровки тонкой призмы, которая учитывает коэффициенты  $s_1$ ,  $s_2$ ,  $s_3$  и  $s_4$ , а также возвращает 12 или более коэффициентов призменной дисторсии
    # cv2.CALIB_FIX_S1_S2_S3_S4 + # Зафиксировать коэффициенты призменной дисторсии и не пытаться подобрать их во время оптимизации
    # cv2.CALIB_TILTED_MODEL + # Использовать модель калибровки наклонного датчика, которая учитывает коэффициенты  $\tau_{ax}$  и  $\tau_{ay}$ , а также возвращает 14 коэффициентов наклонной дисторсии
    # cv2.CALIB_FIX_TAUX_TAUY + # Зафиксировать коэффициенты наклонной дисторсии и не пытаться подобрать их во время оптимизации
    # cv2.CALIB_USE_QR + # Использовать QR-декомпозицию вместо SVD-декомпозиции. Быстрее, но потенциально менее точно
    # cv2.CALIB_FIX_TANGENT_DIST + # Зафиксировать коэффициенты тангенциальной дисторсии и не пытаться подобрать их во время оптимизации
    # cv2.CALIB_FIX_INTRINSIC + # Зафиксировать внутренние параметры видеокамеры и не пытаться подобрать их во время оптимизации
    # cv2.CALIB_SAME_FOCAL_LENGTH + # Обе камеры имеют одинаковое значение фокусного расстояния по осям X и Y
    # cv2.CALIB_ZERO_DISPARITY + # Оптические центры линз обеих видеокамер имеют одинаковые координаты пикселей в выпрямленных видах
    # cv2.CALIB_USE_LU # Использовать LU-декомпозицию вместо SVD-декомпозиции. Гораздо быстрее, но потенциально менее точно
)
```

Листинг 10. Логика расчёта ошибки перепроецирования

```
mean_error = 0

for i in range(len(self.object_points_3D)):
    # Преобразуем каждую точку объекта в соответствующую ей точку на изображении
    image_points, _ = cv2.projectPoints(self.object_points_3D[i], self.rotation_vectors[i], self.translation_vectors[i], self.K, self.D)
    image_points = image_points.reshape(-1, 2) # Приводим к формату N x 2 для соответствия self.image_points_2D[i]

    error = cv2.norm(self.image_points_2D[i], image_points, cv2.NORM_L2) / len(image_points) # Расчитываем абсолютную норму
    mean_error += error

    self.node_logger.info(f'[{self.device_name}] Successfully saved on the path
"../../configs/cameras/{global_settings.USED_CAMERA_MODEL_FOLDER_NAME}/parameters/{self.device_name}.yaml"!\n')
    self.node_logger.info(f'[{self.device_name}] Re-projection Error: {mean_error / len(self.object_points_3D)}\n')
```

Листинг 11. Глобальные параметры решения

```

SIMULATION_TIME_STEP = 32 # Шаг (скорость) симуляции в Webots

# Название папки из ".../surround_view_segbev/configs/cameras/" с характеристиками используемой модели видеокамеры
USED_CAMERA_MODEL_FOLDER_NAME = 'ZED_2'

EGO_VEHICLE_MAX_SPEED = 3.0          # Км/ч
EGO_VEHICLE_MAX_STEERING_ANGLE = 35.0 # Градусы

"""
'Manual' - ручное управление с клавиатуры
'Auto'   - автономное управление алгоритмами
"""

EGO_VEHICLE_CONTROL_MODE = 'Auto'

# Название папок из ".../surround_view_segbev/models/" с весами моделей нейронных сетей глубокого обучения
USED_DETECTOR_FOLDER_NAME = 'YOLO11'

"""
'FastSeg'      - обучена на перспективных изображениях с каждой из 5-ти камер
'FastSegBEV'   - обучена на единых изображениях локальной карты с видом сверху
"""

USED_SEGMENTOR_FOLDER_NAME = 'FastSegBEV'

```

Листинг 12. Параметры Nav2 (начало)

```

amcl:
ros__parameters:
    max_beams: 500 # Используем каждый луч лидара (500 - его горизонтальное разрешение)
    min_particles: 5000
    max_particles: 10000
    update_min_a: 0.05 # Обновлять локализацию каждые 0.05 рад (около 2.8°) поворота
    update_min_d: 0.1 # Обновлять локализацию каждые 10 см движения
    always_reset_initial_pose: true
    scan_topic: /scan_reliable

controller_server:
ros__parameters:
    progress_checker_plugins: ['progress_checker']
    goal_checker_plugins: ['goal_checker']
    controller_plugins: ['FollowPath']
    progress_checker:
        plugin: 'nav2_controller::SimpleProgressChecker'
    goal_checker:
        plugin: 'nav2_controller::SimpleGoalChecker'
        xy_goal_tolerance: 10.0 # Допуск по отклонению в положении (м)
        yaw_goal_tolerance: 3.14 # Допуск по отклонению в ориентации (рад)
    FollowPath:
        plugin: 'nav2_mppi_controller::MPPIController' # 'nav2_regulated_pure_pursuit_controller::RegulatedPurePursuitController'

        # lookahead_dist: 10.0          # Дистанция предсказания траектории движения (м)
        # curvature_lookahead_dist: 0.6 # Дистанция предсказания кривизны траектории (м)

        ## Задействовать curvature_lookahead_dist
        ## вместо зависимости расстояния от скорости
        # use_fixed_curvature_lookahead: true

        # use_rotate_to_heading: false # Разрешить вращение на месте
        # allow_reversing: true       # Разрешить движение задним ходом

```

Листинг 12. Параметры Nav2 (продолжение)

```
motion_model: 'Ackermann'  
AckermannConstraints:  
    min_turning_r: 2.5  
  
batch_size: 2000  
time_steps: 125  
  
vx_min: -1.5  
vx_max: 1.5  
vy_max: 0.0  
wz_max: 0.5  
wz_std: 0.4  
  
visualize: true  
  
prune_distance: 7.5  
  
TrajectoryVisualizer:  
    trajectory_step: 100  
    time_step: 10  
  
critics: [  
    'ConstraintCritic',  
    'GoalAngleCritic',  
    'GoalCritic',  
    'ObstaclesCritic',  
    'CostCritic',  
    'PathAlignCritic',  
    'PathAngleCritic',  
    'PathFollowCritic',  
    'PreferForwardCritic',  
    'VelocityDeadbandCritic',  
]  
  
ObstaclesCritic:  
    consider_footprint: true  
    cost_scaling_factor: 1.5  
    inflation_radius: 3.0  
CostCritic:  
    consider_footprint: true  
    near_goal_distance: 1.0  
PathAlignCritic:  
    cost_weight: 14.0  
    max_path_occupancy_ratio: 0.05  
    trajectory_point_step: 3  
PathAngleCritic:  
    cost_weight: 2.0  
    offset_from_furthest: 4  
    max_angle_to_furthest: 1.0  
PathFollowCritic:  
    offset_from_furthest: 5  
PreferForwardCritic:  
    cost_weight: 0.0  
  
global_costmap:  
global_costmap:  
ros__parameters:  
resolution: 0.05  
track_unknown_space: true
```

Листинг 12. Параметры Nav2 (конец)

```
robot_radius: 0.0
robot_base_frame: 'camera_front_blind' # 'base_link'

# Углы этого-автомобиля: левый задний, левый передний, правый передний, правый задний
footprint: '[ [-1.0, -7.0], [-1.0, 2.0], [1.0, 2.0], [1.0, -7.0] ]' # x, y (camera_front_blind)
# footprint: '[ [-2.0, 1.0], [6.0, 1.0], [6.0, -1.0], [-2.0, -1.0] ]' # y, x (base_link)

plugins: [
    'static_layer',
    'obstacle_layer',
    'inflation_layer',
]

static_layer:
    plugin: 'nav2_costmap_2d::StaticLayer'
    footprint_clearing_enabled: true
obstacle_layer:
    plugin: 'nav2_costmap_2d::ObstacleLayer'
# 0 - частично перезаписывать глобальную карту стоимости каждым валидным наблюдением
# 1 - добавлять препятствия из наблюдения с максимальными значениями параметров ниже
combination_method: 1
observation_sources: scan
scan:
    topic: /scan_reliable
    sensor_frame: lidar
    data_type: 'LaserScan'
    min_obstacle_height: 0.0
    max_obstacle_height: 1.5
    marking: false
# Наносим на карту новые препятствия на расстоянии (м)
    obstacle_min_range: 1.5
    obstacle_max_range: 3.0 # 10.0
    clearing: false
# Очищаем на карте старые препятствия на расстоянии
    raytrace_min_range: 1.5
    raytrace_max_range: 1.5 # 10.0
inflation_layer:
    plugin: 'nav2_costmap_2d::InflationLayer'
    inflation_radius: 3.0
    cost_scaling_factor: 1.5

map_server:
ros_parameters:
    yaml_filename: '../../../../../surround_view_segbev/configs/slam_toolbox/maps/main_wbt/main_wbt.yaml'
```

Листинг 13. Основная часть логики скрипта для настройки конкретной модели камеры

```
# Пока не достигли конца файла с описанием Webots-мира
while line_index != len(webots_world_description_data):
    line = webots_world_description_data[line_index] # Считываем из него очередную строку

    # Если в ней было найдено ключевое слово RangeFinder
    if found_rangefinder_name:
        if 'fieldOfView' in line:
            # Перезаписываем текущее значение на то, что было указано в конфигурационном файле webots_settings.yaml
            webots_world_description_data[line_index] = line.replace(line.split()[1], str(horizontal_fov))
        elif 'width' in line:
            webots_world_description_data[line_index] = line.replace(line.split()[1], str(cameras_image_width))
        elif 'height' in line:
            webots_world_description_data[line_index] = line.replace(line.split()[1], str(cameras_image_height))
        elif 'minRange' in line:
            webots_world_description_data[line_index] = line.replace(line.split()[1], str(min_range))
        elif 'maxRange' in line:
            webots_world_description_data[line_index] = line.replace(line.split()[1], str(max_range))
        elif 'lens' in line:
            lens_node_set_parameters_count = 0
            indentation_level = line.count(' ') # Уровень табуляции в текущей строке

            # Пропускаем строки без интересующих нас параметров
            while '}' not in line:
                lens_node_set_parameters_count += 1 # Считаем количество уже заданных параметров для узла линзы
                line_index += 1
                line = webots_world_description_data[line_index]

            buffering_description_data(line_index) # Буферизуем все строки ниже line_index

            if lens_node_set_parameters_count > 0:
                line_index -= lens_node_set_parameters_count - 1

            # Дописываем или перезаписываем параметры для узла линзы
            webots_world_description_data[line_index] = f'{ " " * indentation_level}center {distortion_center[0]}'
{distortion_center[1]}\n'
            line_index += 1

            webots_world_description_data[line_index] = f'{ " " * indentation_level}radialCoefficients {radial_distortion[0]}'
{radial_distortion[1]}\n'
            line_index += 1

            webots_world_description_data[line_index] = f'{ " " * indentation_level}tangentialCoefficients {tangential_distortion[0]}'
{tangential_distortion[1]}\n'
            line_index += 1
```