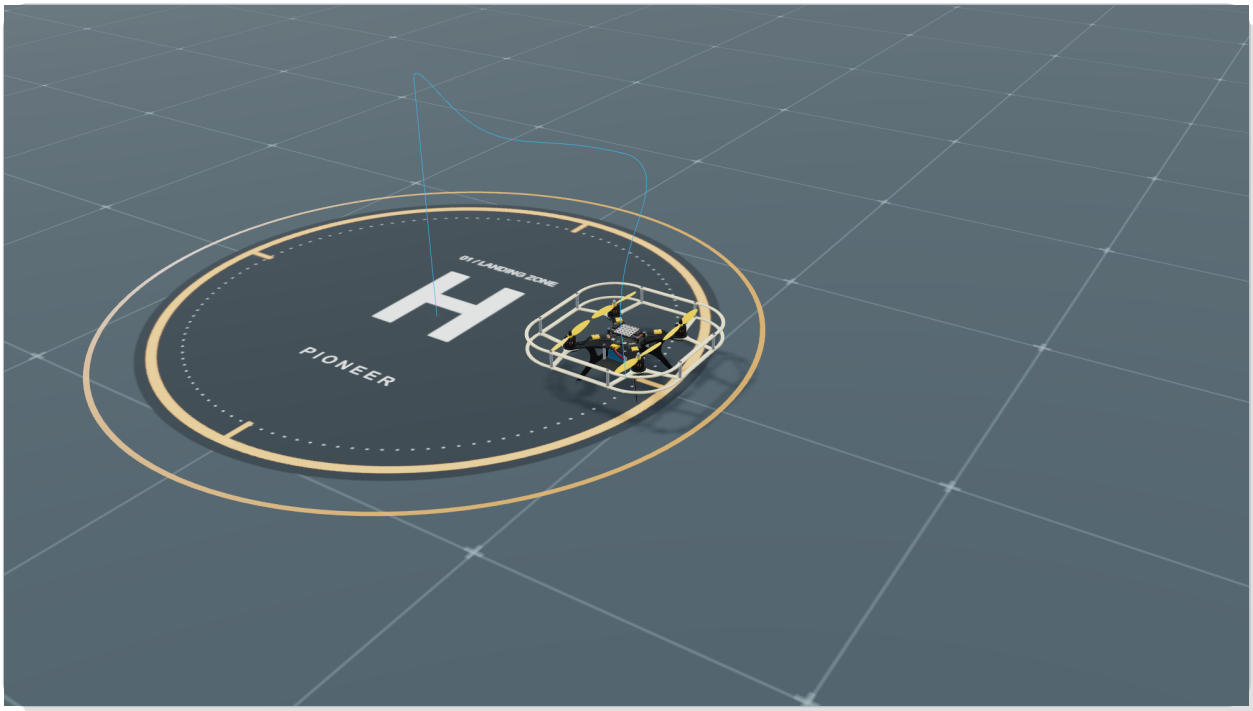
Программирование дронов и создание автономных беспилотных миссий

Курс «Программные средства БПЛА»



Lua **Python** **Blockly** **Симулятор в браузере** **Настоящий дрон**

От первой строки кода до автономной миссии: язык, светодиоды и таймеры, события автопилота, полёт по точкам и фигурам, камера и навигация по маркерам. Каждый пример книги запускается в симуляторе и проверяется автоматически.

Корягин Сергей Андреевич
асп. СПбГЭТУ «ЛЭТИ», педагог ЦО «Кудрово»

г. Кудрово, 2026 simulator.sakoryagin.ru
Версия 1.1.0 · сборка 16.10.2026 08:22

Корягин С. А.

Программирование дронов и создание автономных беспилотных миссий: учебное пособие по курсу «Программные средства БПЛА». — Кудрово, 2026. — 179 с.

Аннотация

Пособие вводит в программирование образовательного квадрокоптера «Геоскан Пионер» на языках Lua и Python. Первая часть — язык Lua, светодиодная индикация, таймеры и событийная модель автопилота, автономные миссии по точкам и параметрическим траекториям. Вторая часть — Python и библиотека `pioneer_sdk`: подключение, полёт, камера, визуальные маркеры ArUco, трекинг и групповой полёт. Третья часть — профессиональная навигация: калибровка камеры, карты маркеров, фильтрация измерений. Все программы запускаются в веб-симуляторе `simulator.sakoryagin.ru` с тем же API, что у настоящего дрона.

Для кого

Для школьников и студентов, начинающих программировать роботов, и педагогов дополнительного образования. Предварительной подготовки по программированию не требуется; математика — в объёме школьной тригонометрии.

Примеры и проверка

Исходный код всех примеров — в папке `examples/` рядом с книгой; каждый файл подписан в тексте. Примеры автоматически прогоняются в симуляторе: взлетают, проходят точки и садятся. Текст примеров можно копировать прямо из PDF.

Версия 1.1.0 · сборка 16.10.2026 08:22

Содержание

1 Введение: как устроен курс 9

2 Подготовка: где писать и запускать код 11

2.1 Два способа запустить программу 11
2.2 Симулятор: что где находится 12
2.3 Редактор кода 13
2.4 Первая программа 14
2.5 Настоящий Пионер 15
2.5.1 Техника безопасности 15
2.5.2 Pioneer Station 16
2.5.3 Python на настоящем дроне 16
2.5.4 Изучение чистого Lua 16

I Программирование на Lua 18

3 Язык Lua 18

3.1 Зачем Lua 18
3.2 Толковый словарь 19
3.3 Переменные и типы 19
3.4 Арифметика и сравнения 21
3.5 Логические операции 23
3.6 Условия 24
3.7 Циклы 25
3.8 Функции 27
3.9 Типичные ошибки и их сообщения 29
3.10 Проверьте себя 30

4 Светодиоды и таймеры 32

4.1 Зачем дрону светодиоды 32
4.2 Толковый словарь 32
4.3 Аппаратная часть: 4 + 25 светодиодов 33
4.4 Модуль Ledbar 34
4.5 Время: почему не sleep 35

4.6 Таймеры . . . 36
4.7 События автопилота . . . 38
4.8 Три способа организовать программу . . . 40
4.8.1 1. Последовательность с sleep . . . 40
4.8.2 2. Таймеры . . . 40
4.8.3 3. Конечный автомат . . . 41
4.9 Примеры комплексных сценариев . . . 42
4.10 Проверьте себя . . . 44
4.11 Практика . . . 45

5 Автономные миссии 48
5.1 Что такое миссия . . . 48
5.2 Толковый словарь . . . 48
5.3 Жизненный цикл миссии . . . 49
5.4 Координаты: где точка . . . 50
5.5 Ступени полёта . . . 51
5.6 Окружность: точки или таймер . . . 56
5.7 Архитектура миссии: конечный автомат . . . 60
5.8 Частые ошибки . . . 67
5.9 Проверьте себя . . . 68
5.10 Практика . . . 69

II Программирование на Python 72

6 Основы Python 72
6.1 Что такое Python и зачем он нужен? . . . 72
6.2 Переменные и типы данных . . . 74
6.3 Арифметические операторы . . . 75
6.4 Логические операции . . . 77
6.5 Ветвление (Условные операторы) . . . 78
6.6 Циклы . . . 79
6.7 Функции . . . 81

7 Pioneer SDK: полёт из Python 83
7.1 Где запускать Python . . . 83

7.2 Подключение и телеметрия . . . 85
7.3 Светодиоды из Python . . . 85
7.4 Первый полёт: моторы, взлёт, посадка . . . 88
7.5 Полёт по точкам . . . 89
7.6 Управление скоростью . . . 93
7.7 Контрольные упражнения с проверкой . . . 94
7.8 Задания к главе . . . 97

8 Камера, компьютерное зрение и рой 99
8.1 Компьютерное зрение: ArUco и AprilTag . . . 99
8.2 Визуальная навигация . . . 109
8.3 Продвинутая работа с камерой . . . 114
8.4 Телеметрия и работа с датчиками . . . 115
8.5 Трекинг человека и жестов . . . 117
8.6 Групповой полет (Рой дронов) . . . 118
8.7 Итоговые проекты курса . . . 120

III Профессиональные навигационные системы 121

9 Теоретические основы визуальной навигации 121
9.1 Введение в ArUco и AprilTag . . . 121
9.2 Принцип работы алгоритма обнаружения . . . 121
9.2.1 1. Предварительная обработка (Preprocessing) . . . 121
9.2.2 2. Поиск контуров (Contour Extraction) . . . 122
9.2.3 3. Исправление перспективы (Perspective Warping) . . . 122
9.2.4 4. Декодирование (Decoding) . . . 122
9.3 Коррекция ошибок и Расстояние Хэмминга . . . 123
9.4 Сравнение типов словарей (Dictionaries) . . . 123
9.5 Математика позиционирования (PnP Problem) . . . 124
9.6 Практические задания . . . 125

10 Создание навигационных карт 127
10.1 Калибровка камеры . . . 127
10.1.1 Процесс калибровки . . . 127
10.2 Генерация и печать маркеров . . . 131

10.2.1 Векторные маркеры 134
10.2.2 Требования к монтажу 139
10.3 Форматы карт (JSON, XML, KML) 139
10.3.1 JSON (JavaScript Object Notation) 147
10.3.2 XML (eXtensible Markup Language) 148
10.3.3 KML (Keyhole Markup Language) 148
10.4 Визуализация и планирование 148

11 Алгоритмы комплексной навигации 150
11.1 Проблема шума и фильтрация 150
11.1.1 Фильтр Калмана (Kalman Filter) 150
11.2 Системы координат 151
11.3 Точная посадка с PID-регулятором 151

12 Техническая реализация программных модулей 153
12.1 Генератор карт (Marker Map Builder) 153
12.1.1 Структура класса Marker 153
12.1.2 Экспорт в XML 153
12.1.3 Экспорт в KML для Google Earth 154
12.2 Модуль визуализации 155

13 Банк практических заданий 156
13.1 Блок 1: Работа с камерой и ArUco (aruco_examples, camera_examples) 156
13.2 Блок 2: Телеметрия (get_telemetry) 156
13.3 Блок 3: Миссии и полет по точкам (mission_examples) . . 157
13.4 Блок 4: Групповой полет (group_flight_examples) 157
13.5 Блок 5: Трекинг объектов (PioneerHumanTracking) 158
13.6 Критерии оценки выполнения 158

14 Типичные ошибки и их устранение 159
14.1 Проблемы с камерой и маркерами 159
14.1.1 Дрон не видит маркер 159
14.1.2 Дрон летит не туда 159
14.2 Проблемы с подключением и Wi-Fi 160
14.2.1 Connection Refused / Timeout 160

14.3 Ошибки в коде Python 160
14.3.1 IndentationError 160
14.3.2 OpenCV Error: Assertion failed 160
14.4 Аварийные ситуации 160
14.4.1 Дрон не реагирует на Land 160

IV Приложения 162

A Справочник Lua API Пионера 162
A.1 Команды автопилота: `ap` 162
A.2 События: `Ev` в `callback(event)` 162
A.3 Таймеры: `Timer` 162
A.4 Светодиоды: `Ledbar` 163
A.5 Датчики: `Sensors` 163
A.6 Идиомы курса 163

B Шпаргалка: Lua и Python 165

C Аппаратура управления 166
C.1 Настройка пульта 166
C.1.1 Настройки вкладки FUNCTION 167
C.1.2 Настройка вкладки SYSTEM 168
C.2 Связь пульта с приемником 169
C.2.1 Процедура привязки 169
C.2.2 Возможные проблемы и их решение 170
C.3 Стики и тумблеры 171
C.3.1 Режимы управления (Тумблеры) 171
C.4 Управление 173
C.4.1 Управление высотой (Газ / Throttle) 173
C.4.2 Вращение по курсу (Рыскание / Yaw) 173
C.4.3 Тангаж (Pitch) — Движение вперед/назад 174
C.4.4 Крен (Roll) — Движение влево/вправо 174

D Методика тестирования и безопасность 176
D.1 Протокол безопасности полетов 176

D.1.1 1. Подготовка зоны (Pre-flight) 176
D.1.2 2. Проверка дрона (Checklist) 176
D.1.3 3. Экстренная ситуация 176
D.2 Тестирование в разных условиях 176
D.2.1 Погодные условия (Ветер) 176
D.2.2 Освещение (Lux) 177

E Готовые конфигурации (Templates) 177
E.1 Файл конфигурации camera_params.yaml 177
E.2 Шаблон скрипта полета (main.py) 177

Список литературы 179

1. Введение: как устроен курс

О чём эта книга

Эта книга учит программировать квадрокоптер «Геоскан Пионер»: от первой строчки на языке Lua до автономной миссии, в которой дрон сам взлетает, облетает заданные точки, реагирует на нештатные ситуации и садится. Вторая половина книги делает то же самое на языке Python, а затем выходит за пределы симулятора — к камере, визуальным маркерам и групповым полётам на настоящем дроне.

Главный принцип курса — **каждый пример запускается**. Все программы из книги лежат в папке `examples/` рядом с текстом и автоматически проверяются в симуляторе: взлетают, пролетают свои точки и садятся. Если пример в книге не совпадает с тем, что вы видите на экране, — это ошибка книги, а не ваша.

Где писать код

Для занятий нужен только браузер: веб-симулятор `https://simulator.sakoryagin.ru` показывает полёт в 3D, умеет Lua, Python и Blockly и использует тот же API, что настоящий Пионер. Настоящий дрон понадобится ближе к концу курса — для камеры и компьютерного зрения. Как запускать программы в симуляторе и на дроне, объясняет глава 2.

Как читать главы

Каждая глава построена одинаково:

1. **Цели** — что вы будете уметь после главы.
2. **Теория с примерами** — короткие программы, которые нужно запустить и сверить результат с комментариями в коде.
3. **Проверьте себя** — вопросы с ответами.
4. **Практика** — задания без готового решения; решения к ним лежат в `examples/lua-tasks/`.
5. **Итоги** — что запомнить.

В тексте встречаются рамки:

- **Информация** — пояснение или ссылка на другой раздел;
- **Внимание** — типичная ошибка или вопрос безопасности;
- **Подсказка** — приём, который экономит время;
- **Правило** — соглашение, которого курс придерживается всегда.

Дорожная карта

1. **Подготовка.** Интерфейс симулятора, первая программа, настоящий Пионер и техника безопасности.
2. **Часть I. Lua.** Язык (переменные, условия, циклы, функции) → светодиоды и таймеры → события автопилота и автономные миссии.
3. **Часть II. Python.** Основы языка → библиотека `pioneer_sdk`: подключение, светодиоды, взлёт, полёт по точкам → камера, маркеры ArUco, трекинг, рой.
4. **Часть III. Профессиональная навигация.** Теория визуальных маркеров, калибровка камеры, карты, фильтрация и точная посадка.
5. **Приложения.** Справочник Lua API, шпаргалка Lua ↔ Python, настройка пульта, протокол безопасности.

Lua и Python в курсе идут в одном порядке: сначала язык, потом светодиоды, потом полёт. Поэтому главу про Python можно читать и отдельно, но полезнее — после Lua: событийная модель автопилота, разобранная на Lua, объясняет, почему Python-программа «ждёт точку» именно так.

Планируемые результаты

После курса вы сможете:

- писать читаемый код на Lua и Python и объяснять, что делает каждая строка;
- управлять бортовой периферией — светодиодами и таймерами — без блокировки программы;
- строить программу полёта как конечный автомат: команда → событие → следующая команда;
- предусматривать аварийные ситуации: удар, разряд батареи, потеря связи;
- переносить программу из симулятора на настоящий дрон без переписывания.

2. Подготовка: где писать и запускать код

Цели главы

- Понять, чем отличаются запуск в симуляторе и на настоящем дроне, и почему код для них один.
- Найти в симуляторе редактор, выбор языка, кнопки запуска и терминал.
- Запустить первую программу и увидеть её вывод.
- Запомнить правила безопасности при работе с настоящим Пионером.

2.1. Два способа запустить программу

У программы для Пионера есть два «исполнителя»:

	Симулятор в браузере	Настоящий Пионер
Что нужно	Любой компьютер с браузером: `https://simulator.sakoryagin.ru`	Дрон, аккумулятор, кабель USB или Wi-Fi, площадка
Lua	Редактор на сайте, язык «**Lua**»	Pioneer Station: загрузка скрипта на борт
Python	Редактор на сайте, язык «**Python**»	`pip install pioneer-sdk`, программа на вашем компьютере
Что видно	Полёт в 3D, терминал, телеметрия	Сам дрон и консоль Pioneer Station
Цена ошибки	Нажать «**Сбросить**»	Сломанный пропеллер или хуже

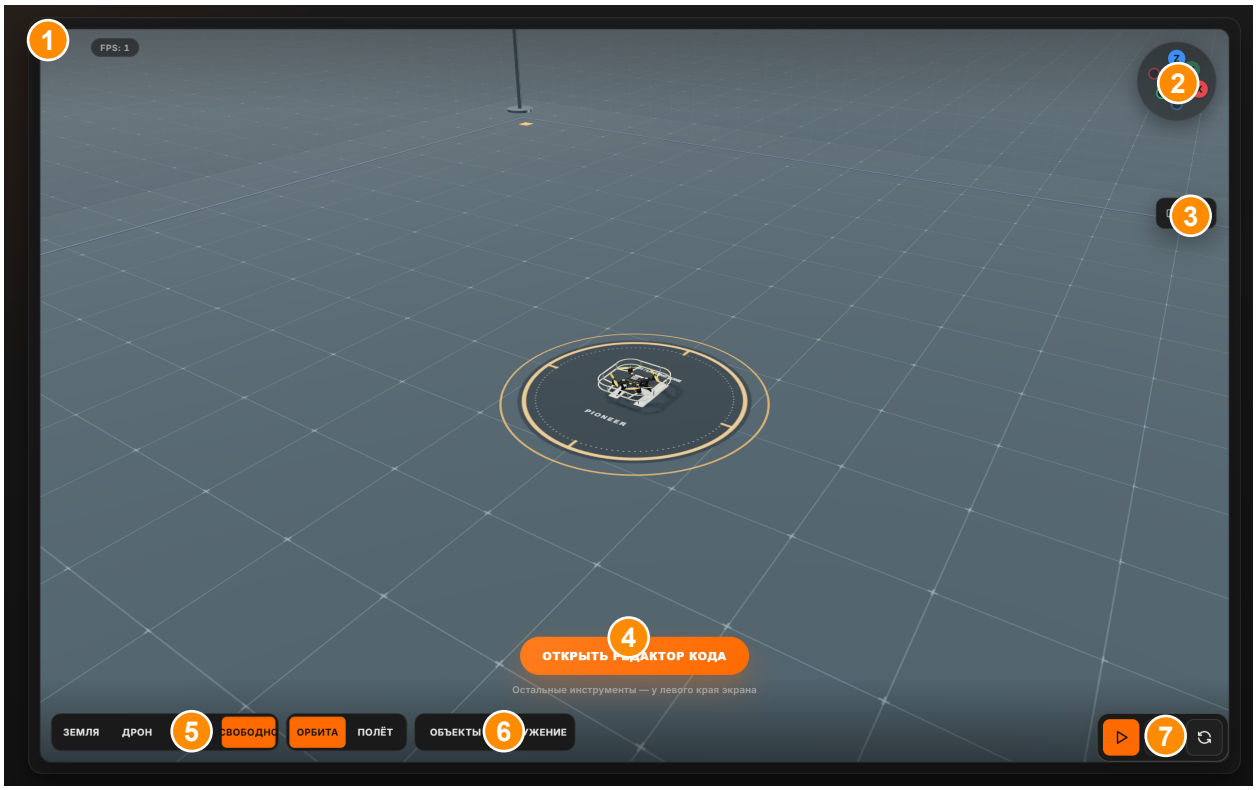
Код в обоих случаях один и тот же: симулятор воспроизводит официальный API автопилота (`ap`, `Ev`, `Timer`, `Ledbar`, `Sensors`) и библиотеку `pioneer_sdk`. Поэтому в курсе мы сначала отлаживаем программу в симуляторе, а на дрон переносим уже проверенную.

Сначала симулятор

Любая новая программа запускается сначала в симуляторе. На настоящем дроне запускается только программа, которая в симуляторе взлетела, выполнила задание и села.

2.2. Симулятор: что где находится

Откройте `https://simulator.sakoryagin.ru`. При первом запуске появится короткая экскурсия — её можно пройти или пропустить. После неё экран выглядит так:

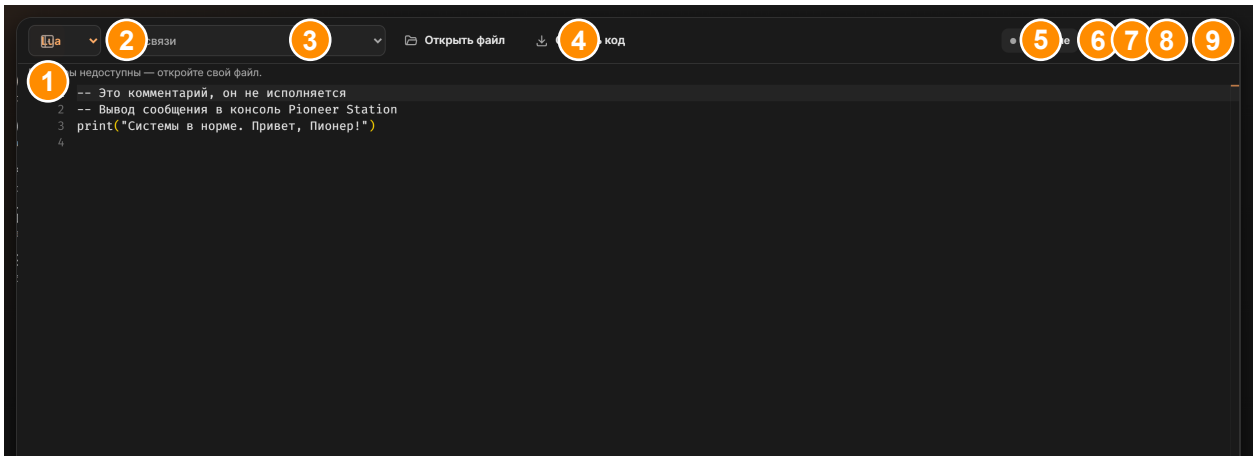


1. Кнопка панелей. За ней прячется «рельс» с панелями: «**Код**», «**Полёт**» (телеметрия), «**Рой**», «**Сцена**», «**Учебник**», «**Пульт**», «**Настройки**».
2. Компас осей. Ось **X** красная, **Y** зелёная, **Z** синяя и смотрит вверх. Щелчок по оси поворачивает камеру вдоль неё.
3. Окна камер дрона: передняя и нижняя камеры, их же «видит» программа.
4. «**Открыть редактор кода**» — сюда пишется программа. Позже эта кнопка исчезнет, редактор открывается панелью «**Код**».
5. Камера: «**Земля**» (вид сверху), «**Дрон**» (камера следует за дроном), «**FPV**» (от лица дрона), «**Свободно**» (орбита вокруг точки или свободный полёт камеры).
6. «**Объекты**» и «**Окружение**»: расстановка домов, маркеров, площадок; время суток и погода.
7. «**Запустить**», «**Стоп**» и «**Сбросить**». Сброс возвращает дрон на стартовую площадку и очищает след полёта.

Камеру вращают левой кнопкой мыши, сдвигают правой, приближают колёсиком.

2.3. Редактор кода

Нажмите «**Открыть редактор кода**». Редактор занимает экран целиком, сцена при этом продолжает жить: программа работает и после закрытия редактора.



1. Рельс панелей: «**Код**» (этот редактор), «**Полёт**», «**Рой**», «**Сцена**», «**Учебник**»; внизу — тема, «**Пульт**», «**Настройки**».
2. Язык программы: «**Lua**», «**Python**» или «**Blockly**». Переключение не стирает код: у каждого языка своя вкладка.
3. Готовые примеры — в том числе все программы из этой книги (папки `04_language`, `05_leds`, `06_missions`).
4. «**Открыть файл**» с компьютера и «**Скачать код**» в файл.
5. Состояние дрона: «**На земле**», «**Взлёт**», «**В полёте**», «**Посадка**».
6. Справочник API: описание и пример для каждой функции.
7. Терминал: сюда попадает вывод `print` и сообщения автопилота.
8. Уведомления: перед запуском симулятор проверяет программу и объясняет найденные ошибки здесь.
9. Закрыть редактор и вернуться к сцене. Кнопки «**Запустить**», «**Стоп**», «**Сбросить**» — внизу справа, как и на сцене.

Пять шагов до запуска

Открыть редактор → выбрать язык → вставить или набрать программу → «**Запустить**» → закрыть редактор и смотреть на сцену. Если дрон ведёт себя не так, как ожидалось, — «**Стоп**», «**Сбросить**», и в терминале читать, что произошло.

2.4. Первая программа

Традиционно первая программа печатает приветствие. На дроне нет экрана, но есть консоль: в симуляторе это терминал редактора, в Pioneer Station — окно «Консоль».

lua/02_sensors/getting_started_hello.lua

```lua
-- Это комментарий, он не исполняется
-- Вывод сообщения в консоль Pioneer Station
print("Системы в норме. Привет, Пионер!")
```

Выберите язык «**Lua**», вставьте программу и нажмите «**Запустить**». Затем откройте терминал (значок `>_` в заголовке редактора):

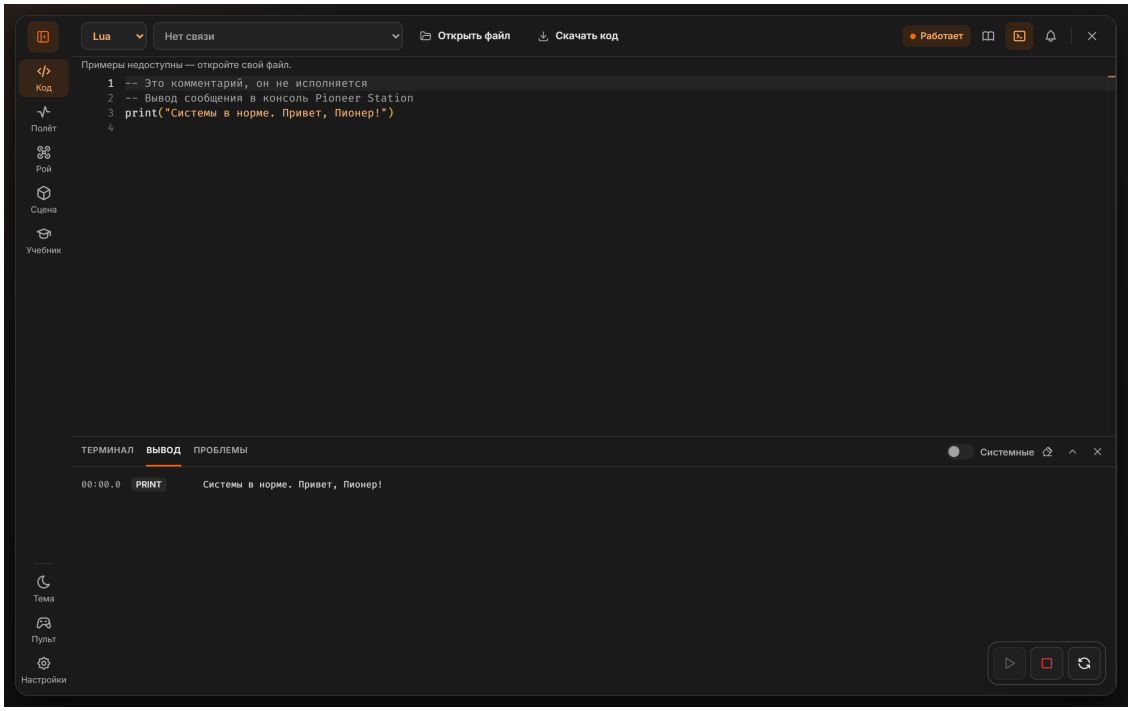


Рис. 1: Терминал показывает всё, что программа напечатала через `print`, и сообщения автопилота.

Что здесь произошло:

- строки, начинающиеся с `--`, — комментарии, Lua их не выполняет;
- `print(…)` печатает текст в консоль. Текст — это **строка**, она записывается в кавычках;
- программа закончилась сразу после печати. Дрон не взлетел, потому что мы не дали ему ни одной команды.

Задание 2.1: Проверка связи

Измените текст в кавычках на своё имя и запустите программу ещё раз. Затем добавьте вторую строку `print(2 + 2)` и убедитесь, что Lua умеет считать.

Где лежат примеры

Каждая программа в книге подписана именем файла, например `lua/02_sensors/getting_started_hello.lua`. Это путь внутри папки `examples/` курса. Те же файлы доступны в редакторе симулятора через список примеров, а текст программы можно скопировать прямо из PDF — отступы и дефисы при копировании сохраняются.

2.5. Настоящий Пионер

Всё, что отлажено в симуляторе, переносится на дрон. Но дрон — электромеханическое устройство с вращающимися пропеллерами, поэтому порядок работы с ним строже.

2.5.1. Техника безопасности

Критически важно

Пренебрежение этими правилами приводит к травмам и поломке оборудования. Они не обсуждаются.

- **Механика — только без питания.** Любые монтажные работы, замена деталей, подключение модулей — на обесточенном аппарате.
- **Отладка — без пропеллеров.** При загрузке и проверке скриптов, тестировании моторов и настройке пульта пропеллеры должны быть сняты. Случайный запуск моторов с пропеллерами — неконтролируемый взлёт.
- **Аккумуляторы.** Только исправные LiPo-батареи, зарядка под присмотром. Вздувшийся или повреждённый аккумулятор не используется.

- **Порядок включения.** Сначала пульт, потом питание дрона. Выключение — в обратном порядке.
- **Рабочее место.** На столе нет лишних предметов и проводов; с пропеллерами дрон создаёт сильный поток воздуха.

2.5.2. Pioneer Station

Основная среда для Lua-программ на настоящем дроне — **Pioneer Station**: редактор с подсветкой синтаксиса, загрузчик скриптов на борт, консоль для `print` и телеметрии, менеджер прошивок. Дистрибутив бесплатно доступен на сайте поддержки Геоскан.

Порядок подключения:

1. Убедитесь, что пропеллеры сняты (если вы не планируете немедленный вылет на полигоне).
2. Подключите аккумулятор. Прозвучит мелодия инициализации, светодиоды начнут мигать.
3. Соедините дрон с компьютером кабелем micro-USB.
4. Запустите Pioneer Station и выберите COM-порт (обычно определяется сам).
5. Вставьте программу в редактор и нажмите «Загрузить». Скрипт передаётся в память автопилота и запускается; результат — в окне «Консоль».

2.5.3. Python на настоящем дроне

Python-программа работает не на борту, а на вашем компьютере, и общается с дроном по Wi-Fi через библиотеку `pioneer_sdk`. Установка и подключение разобраны в главе 7. Ту же программу можно направить и на симулятор — тогда по Wi-Fi полетит виртуальный дрон на сайте (раздел 7.1).

2.5.4. Изучение чистого Lua

Синтаксис Lua можно тренировать и без дрона: симулятор для этого подходит (терминал показывает `print`), а на компьютере удобны ZeroBrane Studio, Visual Studio Code с расширением Lua или онлайн-интерпретаторы. Код оттуда для дрона всё равно переносится в симулятор или Pioneer Station.

Итоги

- Программа для Пионера одна, исполнителей два: симулятор в браузере и настоящий дрон.
- В симуляторе: редактор → язык → «**Запустить**»; вывод — в терминале, ошибки — в уведомлениях.
- На дроне: безопасность, Pioneer Station для Lua, `pioneer_sdk` для Python.

Часть I.

Программирование на Lua

3. Язык Lua

Цели главы

- Различать типы данных Lua и объявлять переменные — локальные и глобальные.
- Записывать арифметику, сравнения и логические условия.
- Управлять ходом программы: ветвления `if`, циклы `for`, `while`, `repeat`.
- Писать свои функции и понимать, как автопилот вызывает нашу функцию `callback`.
- Узнавать типичные ошибки новичка по сообщению в терминале.

3.1. Зачем Lua

Lua — лёгкий встраиваемый язык, созданный в начале 1990-х для расширения программ скриптами. Он компактен, не требует много памяти и быстро работает, поэтому живёт внутри игр, роботов и встраиваемых систем. Автопилот Пионера исполняет Lua прямо на борту: программа загружается в память контроллера и работает без компьютера.

В этом курсе Lua нужен, чтобы:

- управлять светодиодами через модуль `Ledbar` (глава 4);
- выполнять действия по расписанию через `Timer`, не останавливая программу;
- отвечать на события автопилота — взлёт завершён, точка достигнута, батарея садится — в функции `callback(event)` (глава 5).

Сейчас запоминать эти имена не нужно: в этой главе мы работаем только с языком.

Как работать с главой

Каждый пример запускайте в симуляторе (язык «**Lua**», кнопка «**Запустить**») и сверяйте вывод в терминале с комментариями в коде: справа от `print` написано, что должно появиться.

3.2. Толковый словарь

Число вещественное или целое значение: `3.14`, `10`; математика — в библиотеке `math`.

Строка текст в кавычках: `"Lua"`; склейка — оператор `..`, длина — `#s`.

Булево `true` или `false`; результат сравнения и материал для условий.

nil «ничего»: переменная без значения, отсутствующее поле таблицы.

Таблица единственная составная структура Lua: массив `t[1]` и словарь `t.x` одновременно.

Функция именованный блок кода с параметрами и результатом.

Область видимости `local` делает переменную видимой только внутри текущего блока; без `local` переменная глобальная — видна во всём скрипте.

Истинность ложными считаются только `false` и `nil`; ноль и пустая строка — истина.

ipairs / pairs обход таблицы: `ipairs` — по номерам 1..N, `pairs` — по всем ключам в произвольном порядке.

table.unpack раскладывает таблицу в список значений: `table.unpack({1,0,0})` даёт `1, 0, 0`. Часто пишут `local unpack = table.unpack`.

3.3. Переменные и типы

Переменная — это имя, за которым закреплено значение. Тип значения определяется автоматически: число, строка, булево, таблица, функция или nil. Узнать тип можно функцией `type()`.

lua/04_language/variables_intro.lua

```lua
-- локальная числовая переменная
local x = 10
-- глобальная строковая переменная
```

```
name = "Pioneer"
-- локальная булева переменная
local ready = true
-- таблица с позиционными элементами и именованным полем
local t = {1, 2, 3, speed = 3.5}
-- множественное присваивание
x, y = 1, 2
-- обмен значениями
x, y = y, x
-- выясняем типы значений
print(type(x), type(name), type(ready), type(t))  -- number string
↪  boolean table
```

Что происходит.

- `local x = 10` — локальная переменная: живёт в том блоке, где объявлена.
- `name = "Pioneer"` без `local` — глобальная: её видно из любой функции скрипта. Так в курсе объявляют таймеры (глава 4), которые должны жить до конца программы.
- `x, y = 1, 2` присваивает две переменные сразу, а `x, y = y, x` меняет их местами без третьей переменной.
- `print(type(x), ...)` печатает имена типов через табуляцию.

Таблицы

Таблица — и массив, и словарь. Позиционные элементы нумеруются **с единицы**, именованные поля доступны через точку.

lua/04_language/lua_variables_tables.lua

```
-- словарь (именованные поля)
local p = {x = 1, y = 2}
-- массив (позиционные элементы)
local a = {10, 20, 30}
-- добавление нового поля
p.z = 3
-- добавление нового элемента массива
a[4] = 40
-- поле словаря, четвёртый элемент и длина массива
print(p.z, a[4], #a)  -- 3  40  4
```

- `#a` — длина массива. После `a[4] = 40` она равна 4.
- Присваивание `t.speed = nil` удаляет поле из таблицы.
- Таблицами в курсе задаются цвета `{1, 0, 0}` и точки маршрута `{x, y, z}`.

local по умолчанию

Объявляйте переменные через `local`. Глобальными оставляйте только то, что должно жить всю программу: таймеры и обязательную функцию `callback` (о ней — в разделе «Функции» этой главы). Переменная, объявленная через `local` в самом начале файла, тоже живёт всю программу и видна всем функциям ниже, — так в миссиях хранят текущее состояние (глава 5).

Задачи для самостоятельного решения

1. Объявите локальную переменную `x` со значением `42` и напечатайте её.
2. Создайте таблицу `t` с полем `speed = 3.5` и элементами `1, 2, 3`; напечатайте `t.speed` и `#t`.
3. Добавьте в `t` поле `mode = "auto"`, затем удалите поле `speed`.
4. Создайте таблицу `color = {r = 1, g = 0, b = 0}` и напечатайте её поля одной командой `print`.
5. Объявите `local a, b = 1, 2`, поменяйте значения местами одной строкой и напечатайте результат.
6. Напечатайте `type(nil)`, `type("5")` и `type(5)`. Чем отличаются два последних?

3.4. Арифметика и сравнения

Операции: `+ - * / %` (остаток от деления) и `^` (степень). Сравнения: `< <= > >= ==` и `~=` («не равно»). Приоритет как в математике: сначала `^`, затем `* / %`, потом `+ -`; скобки меняют порядок.

lua/04_language/lua_arithmetic_ops.lua

```lua
-- приоритет: умножение выполняется раньше сложения
local a = 2 + 3 * 4
-- остаток от деления (modulo)
local r = 5 % 2
-- возведение 2 в степень 3 (результат всегда вещественный)
local p = 2 ^ 3
print(a, r, p)        -- 14  1  8.0
-- сравнения дают булево значение
local eq = (a == 14)
local gt = (a > 10)
```

```lua
print(eq, gt)          -- true  true
-- "целочисленное" деление с округлением вниз
local q = math.floor(17 / 4)
-- евклидова длина вектора
local len = math.sqrt(3*3 + 4*4)
print(q, len)          -- 4  5.0
```

Что происходит.

- `2 ^ 3` даёт `8.0`, а не `8`: степень и деление `/` всегда возвращают вещественное число. Целые и вещественные числа в Lua 5.3 — два подвида одного типа, в арифметике они смешиваются свободно.
- Оператора `**` в Lua нет — это Python. Степень только через `^`.
- `math.floor(17 / 4)` — целочисленное деление с округлением вниз. В симуляторе работает и `//`, но `math.floor` переносимее.
- `math.sqrt`, `math.abs`, `math.max`, `math.min`, `math.pi` — то, что понадобится в миссиях. Тригонометрия Lua и автопилот измеряют углы в **радианах**: полный оборот — $2\pi \approx 6.28$ рад, $180^\circ = \pi$. Перевод градусов в радианы: `math.rad(deg)`, обратно — `math.deg(rad)`.

Ограничить значение диапазоном

`v = math.max(0, math.min(1, v))` «зажимает» `v` между 0 и 1. Этот приём пригодится для яркости светодиодов.

Задачи для самостоятельного решения

1. Напечатайте `3 + 4 * 2 - 5` и `(3 + 4) * (2 - 5)`.
2. Напечатайте остаток `17 % 4` и проверьте условием, чётно ли число `n = 10`.
3. Вычислите длину вектора `(x, y) = (3, 4)` через `math.sqrt(x*x + y*y)`.
4. Переведите `90` градусов в радианы двумя способами: `deg * math.pi / 180` и `math.rad(deg)`. Сравните результаты.
5. Реализуйте линейную интерполяцию `lerp = a + t * (b - a)` для `a = 0`, `b = 10`, `t = 0.25`.
6. Напечатайте среднее трёх чисел `(a + b + c) / 3`.

3.5. Логические операции

Логика: `and` (И), `or` (ИЛИ), `not` (НЕ). Ложными считаются только `false` и `nil` (истинность): число `0` и пустая строка — истина. Операторы `and` и `or` вычисляют правую часть только при необходимости («короткое замыкание»).

lua/04_language/lua_logic_ops.lua

```lua
local a = 42
-- сравнение возвращает булево значение
local gt = a > 10
-- логическое И и НЕ
local ok = true and not false
-- "тернарный" идиом в Lua через and/or
local sel = gt and 1 or 0
-- композиция условий "в диапазоне"
local within = (a >= 0) and (a <= 100)
print(ok, sel, within)        -- true  1  true

local name = "Pioneer"
-- проверка непустой строки
local nonempty = (name ~= nil) and (name ~= "")
-- исключающее ИЛИ для булевых
local p, q = true, false
local xor = (p and not q) or (q and not p)
print(nonempty, xor)          -- true  true
```

Что происходит.

- `gt and 1 or 0` — идиома «если `gt`, то 1, иначе 0». В Lua нет тернарного оператора, его заменяет эта пара.
- `(a >= 0) and (a <= 100)` — проверка диапазона. Запись `0 <= a <= 100` в Lua не работает.
- `name ~= nil and name ~= ""` — сначала убеждаемся, что значение есть, и только потом сравниваем со строкой. Так защищаются от `nil`.
- Строки сравниваются лексикографически: `"a" < "b"` истинно.

Задачи для самостоятельного решения

1. Для `voltage = 3.4` и `temp = 85` напечатайте «сигнал опасности»: `voltage < 3.5 or temp > 80`.
2. Напишите предикат «индекс чётный» для `i = 7`: `i % 2 == 0`.
3. Выберите максимум из `a` и `b` одной строкой через `and/or`.
4. Инвертируйте булеву переменную `on` и напечатайте её до и после.

5. Проверьте закон де Моргана: напечатайте `not (p and q)` и `(not p) or (not q)` для всех четырёх сочетаний `p`, `q`.
6. Напишите условие «`t` не `nil` и у `t` есть поле `value`».

3.6. Условия

Ветвление записывается словами `if` … `then` … `elseif` … `then` … `else` … `end`. Ветка `elseif` может повторяться, `else` — не обязательна, `end` — обязателен всегда.

lua/04_language/lua_conditions_if.lua

```lua
-- классификация числа x: каскад if / elseif / else
local function classify(x)
  if x > 0 then
    return "positive"
  elseif x == 0 then
    return "zero"
  else
    return "negative"
  end
end

print(classify(5))    -- positive
print(classify(0))    -- zero
print(classify(-3))   -- negative
```

Что происходит. Функция `classify` проверяет условия по порядку и возвращает первое подходящее слово. Для `0` первая проверка `x > 0` ложна, вторая `x == 0` истинна — получаем `"zero"`.

Охранное условие

Обращение к полю несуществующей таблицы — ошибка «attempt to index a nil value». Защищаются от неё проверкой перед обращением:

lua/04_language/lua_conditions_check.lua

```lua
-- "охранное" условие: работаем с полем, только если таблица и поле
↪  существуют
local function doubled(t)
  if t and t.value then
    return t.value * 2
  end
```

```
  return "нет данных"
end

print(doubled({value = 42}))  -- 84
print(doubled(nil))           -- нет данных
```

Условие `t and t.value` читается так: «t существует, и у неё есть поле `value`». Если `t` равно `nil`, до `t.value` дело не доходит — спасает короткое замыкание `and`.

Таблица вместо каскада if

Когда вариантов много и каждому соответствует действие, удобнее таблица: `colors[state]` или даже таблица функций `actions[state]()`. Так в главе 4 устроен светофор — самый маленький конечный автомат.

Задачи для самостоятельного решения

1. Напишите функцию, которая для напряжения `v` возвращает `"low"` при `v < 3.7` и `"ok"` иначе. Проверьте на 3.5 и 4.1.
2. Присвойте `parity` значение `"even"` при чётном `n` и `"odd"` при нечётном.
3. Тройное ветвление по температуре: `temp < 0` → `"cold"`, `0 <= temp < 20` → `"cool"`, иначе `"warm"`.
4. Напишите ветвление, выбирающее цвет: `{1,0,0}` для `state == "RED"`, `{1,1,0}` для `"YELLOW"`, иначе `{0,1,0}`.
5. Защитите деление от нуля: если `b ~= 0`, напечатайте `a / b`, иначе — сообщение об ошибке.
6. Если `count == 0`, присвойте `count = 1`, иначе увеличьте на 1.

3.7. Циклы

Три вида повторения: числовой `for` (счётчик от начала до конца включительно, с шагом), `while` (пока условие истинно) и `repeat ... until` (тело выполняется хотя бы один раз). Для таблиц есть `for ... in ipairs(a)` и `for ... in pairs(t)`.

lua/04_language/loops_sum.lua

```lua
-- суммирование 1..10 числовым циклом for
local sum = 0
for i = 1, 10 do
  sum = sum + i
end
print(sum)  -- 55
```

Границы `for i = 1, 10` включительны: `i` принимает значения от 1 до 10. Светодиоды линейки (29 штук, глава 4) нумеруются с нуля, поэтому их обходят циклом `for i = 0, 28 do`.

lua/04_language/loops_while.lua

```lua
-- while: увеличиваем x, пока он меньше 5
local x = 0
while x < 5 do
  x = x + 1
end
print(x)  -- 5

-- repeat ... until: тело выполняется минимум один раз
repeat
  x = x - 1
until x == 0
print(x)  -- 0
```

lua/04_language/loops_iterators.lua

```lua
-- ipairs: обход массива по порядку, умножаем элементы на 2
local a = {10, 20, 30}
for i, v in ipairs(a) do
  a[i] = v * 2
end
print(a[1], a[2], a[3])  -- 20  40  60

-- pairs: обход словаря, увеличиваем значения на 1
local t = {x = 1, y = 2}
for k, v in pairs(t) do
  t[k] = v + 1
end
print(t.x, t.y)          -- 2  3
```

Что происходит.

- `ipairs` перебирает массив по порядку и останавливается на первом «пропуске» (`nil`); `pairs` перебирает все ключи словаря, порядок не гарантирован.

- `break` прерывает цикл. Оператора `continue` в Lua нет — вместо него условие внутри тела.

Вечный цикл останавливает дрон

Цикл `while true do ... end` без выхода никогда не вернёт управление автопилоту: события перестанут приходить, таймеры — срабатывать. Повторяющиеся действия на дроне делают таймером `Timer.new` (глава 4), а не бесконечным циклом. Симулятор не даст запустить такую программу и объяснит почему.

lua/04_language/lua_platform_infinite_loop.lua

```lua
-- ОПАСНО! Этот код "повесит" выполнение скрипта
while true do
    print("Я завис")
end
```

Задачи для самостоятельного решения

1. Посчитайте сумму чисел от 1 до 100.
2. Найдите максимум в массиве `a = {3, 9, 2, 7}`.
3. Постройте массив `b` из квадратов элементов `a`.
4. Вычислите факториал `n = 6` циклом `while`.
5. Заполните таблицу `idx` значениями от 0 до 28 (как номера светодиодов) и напечатайте `#idx`.
6. Постройте массив из первых 10 чисел Фибоначчи.
7. Скопируйте словарь `t = {x = 1, y = 2}` в `copy`, увеличив каждое значение на 1.

3.8. Функции

Функция — именованный блок кода, который принимает параметры и возвращает результат. Объявляется словом `function`, заканчивается `end`, результат отдаёт `return`.

lua/04_language/lua_functions.lua

```lua
-- функция с параметрами и одним результатом
local function average(a, b)
  return (a + b) / 2
end
print(average(1, 2))  -- 1.5
print(average(1, 3))  -- 2.0

-- функция может вернуть несколько значений сразу
local function describe(x)
  local color = (x > 0) and "green" or "red"
  local word = (x == 0) and "zero" or "nonzero"
  return color, word
end
print(describe(0))    -- red  zero
```

Что происходит.

- `local function average(a, b)` — локальная функция: видна только в этом файле ниже объявления. Вызов `average(1, 2)` подставляет `a = 1`, `b = 2` и возвращает `1.5`.
- Функция может вернуть несколько значений: `return color, word`. Так устроены датчики дрона: `local x, y, z = Sensors.lpsPosition()` возвращает три координаты сразу (все датчики — в приложении A).
- Функцию можно передать как значение. Например, `Timer.callLater(3, blink)` просит вызвать функцию `blink` через 3 секунды (таймеры разобраны в главе 4): сюда передаётся сама функция `blink`, а `Timer.callLater(3, blink())` передал бы результат её вызова. Второе — ошибка, симулятор её ловит.

Функция, которую вызываем не мы

Обычно функцию вызывает наша программа. Но есть функция, которую вызывает **автопилот**:

Обработчик событий

```lua
function callback(event)
  -- сюда автопилот передаёт событие:
  -- взлёт завершён, точка достигнута, дрон сел...
end
```

Мы только определяем её — глобально, без `local`, ровно с таким именем. Когда произойдёт событие, автопилот сам вызовет `callback` и передаст номер события. Что с ним делать, разбирают главы 4 и 5; пока запомните: **в каждой программе, которая командует автопилотом, есть `callback`**, даже пустой. Программам, которые только зажигают светодиоды, он не нужен.

Задачи для самостоятельного решения

1. Напишите функцию `square(x)`, возвращающую квадрат числа; проверьте на 5.
2. Напишите `isEven(n)`, возвращающую `true` для чётных чисел.
3. Напишите `clamp(v, lo, hi)`, ограничивающую `v` диапазоном `[lo, hi]`.
4. Напишите `toRadians(deg)` и проверьте, что `toRadians(180)` равно `math.pi`.
5. Напишите `polar(r, deg)`, возвращающую две координаты `x, y` точки окружности радиуса `r` под углом `deg` градусов.
6. Напишите функцию `describeBattery(v)`: ниже 3.5 В — `"critical"`, ниже 3.8 — `"low"`, иначе `"ok"`.

3.9. Типичные ошибки и их сообщения

Терминал симулятора и консоль Pioneer Station показывают текст ошибки и номер строки. Самые частые:

Сообщение	Причина и лечение
`attempt to call a nil value (global 'Print')`	Регистр букв: `Print`, `PRINT` и `print` — три разных имени. Все функции языка пишутся строчными; с заглавной — только модули `Ledbar`, `Timer`, `Sensors`, `Ev`.
`attempt to index a nil value`	Обращение к полю переменной, в которой `nil`: таблица не создана или опечатка в имени. Проверьте объявление и используйте охранное условие.
`attempt to compare nil with number`	Переменная не получила значения до сравнения — например, опечатка в имени или `local` внутри другого блока.
`'end' expected`	Не хватает `end`: у каждого `if`, `for`, `while`, `function` свой `end`. Выравнивайте код отступами — так пропуск виден.
`unexpected symbol near '**'`	Степень в Lua — `^`, а не `**`.

⚠ Индексы: с нуля или с единицы

Таблицы Lua нумеруются с **1**: `points[1]` — первая точка. Светодиоды — аппаратный модуль, они нумеруются с **0**: `leds:set(0, ...)` — первый светодиод. Путаница между ними — самая частая ошибка следующей главы.

3.10. Проверьте себя

1. Что напечатает `print(10 / 2)`?
   - **(a)** `5`
   - **(b)** `5.0`
   - **(c)** `5,0`

   Ответ: (b) — деление всегда даёт вещественное число.
2. Какое из значений ложно в условии `if`?
   - **(a)** `0`
   - **(b)** `""`
   - **(c)** `nil`

Ответ: (c) — ложны только `false` и `nil`.

3. Чему равно `#{10, 20, 30}`?

   (a) `2`

   (b) `3`

   (c) `30`

   Ответ: (b).

4. Сколько раз выполнится тело `for i = 0, 28 do`?

   (a) 28

   (b) 29

   (c) 27

   Ответ: (b) — обе границы включительно.

5. Какой из вызовов передаёт функцию, а не результат её вызова?

   (a) `Timer.callLater(2, blink())`

   (b) `Timer.callLater(2, blink)`

   Ответ: (b).

6. Что делает `x, y = y, x`?

   (a) Ошибка

   (b) Меняет значения местами

   (c) Присваивает обеим `y`

   Ответ: (b).

Итоги главы

- Переменные объявляются через `local`; глобальные — только для долгоживущего состояния.
- Таблица — массив (с 1) и словарь одновременно; цвета и точки маршрута — таблицы.
- Ложны только `false` и `nil`; `cond and A or B` заменяет тернарный оператор.
- Циклы `for`, `while`, `repeat`; бесконечный цикл на дроне недопустим — вместо него таймер.
- Функции возвращают несколько значений и передаются как аргументы; `callback(event)` вызывает автопилот.

4. Светодиоды и таймеры

Цели главы

- Понять, как устроена светодиодная линейка Пионера и почему её адреса — `0..28`.
- Задавать цвет через модуль `Ledbar` и модель RGB.
- Делать анимации на таймерах `Timer.new` и `Timer.callLater`, не останавливая программу.
- Реагировать на события автопилота в `callback(event)`.
- Строить индикацию как конечный автомат: состояние → событие → переход.

4.1. Зачем дрону светодиоды

Светодиоды — первый видимый результат программы: дрон стоит на земле, а цвет меняется. Поэтому с них начинают. Но это ещё и рабочий инструмент: по цвету понятно, в каком состоянии программа — готовится, летит, садится, сообщает об ошибке, — даже когда терминала нет рядом. В главе 5 мы будем подсвечивать этапы миссии именно так.

В этой главе дрон не взлетает. Все примеры безопасно запускать и в симуляторе, и на настоящем Пионере со снятыми пропеллерами.

4.2. Толковый словарь

Ledbar модуль управления RGB-светодиодами: `Ledbar.new(N)` создаёт объект, `leds:set(i, r, g, b)` задаёт цвет.

RGB три компоненты цвета — красная, зелёная, синяя — числа от `0.0` (выключено) до `1.0` (максимум).

Индекс светодиода номер в линейке, от `0` до `N-1`.

Таймер «будильник», который вызывает функцию через заданное время (`Timer.callLater`) или периодически (`Timer.new`).

Тик один вызов функции таймера.

Событие сигнал от автопилота — взлёт завершён, точка достигнута, батарея разряжена. Передаётся в `callback(event)` как значение `Ev.*`.

Состояние текущий режим логики программы, например `"RED"` или

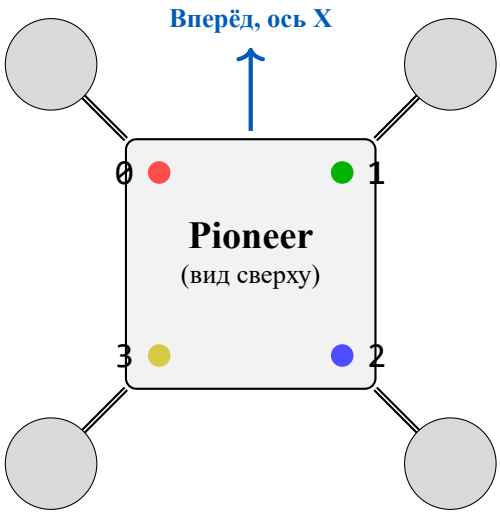
`"FLIGHT"`; хранится в переменной.

Конечный автомат программа, которая всегда находится ровно в одном состоянии и переходит в другое только по событию.

Блокирующий вызов функция, которая останавливает всю программу, пока не завершится, например `sleep(t)`.

4.3. Аппаратная часть: 4 + 25 светодиодов

На плате Пионера четыре RGB-светодиода — по одному у каждого луча. Модуль светодиодной матрицы добавляет ещё 25. Программа видит их как одну линейку: индексы `0..3` — светодиоды платы, `4..28` — модуль. Всего 29, поэтому в курсе линейка всегда создаётся как `Ledbar.new(29)`: на дроне без модуля лишние индексы просто ни на что не влияют, а в симуляторе отображаются все 29.



Индексы `0..3` — плата; `4..28` — модуль матрицы.
Какой индекс какому углу соответствует, проверяет задание ниже.

Модель RGB

Цвет — три числа от `0.0` до `1.0`: красный, зелёный, синий. Смешивая каналы, получаем любой оттенок:

`{1, 0, 0}` красный	`{0, 1, 0}` зелёный	`{0, 0, 1}` синий	`{1, 1, 0}` жёлтый
`{1, 0, 1}` фиолетовый	`{0, 1, 1}` бирюзовый	`{1, 1, 1}` белый	`{0, 0, 0}` выключен

Значения меньше единицы — тусклее: `{0.2, 0, 0}` — тёмно-красный. В Python светодиодами управляет метод `led_control` библиотеки

`pioneer_sdk`, и компоненты там — целые числа `0..255` (глава 7); в Lua — только `0..1`.

4.4. Модуль Ledbar

Две функции, которых достаточно

- `local leds = Ledbar.new(29)` — создать объект линейки из 29 светодиодов. Вызывается один раз в начале программы.
- `leds:set(i, r, g, b)` — задать цвет светодиода `i` (от `0` до `28`). Обратите внимание на двоеточие: `set` — метод объекта `leds`.

lua/05_leds/leds_basic_green.lua

```lua
local unpack = table.unpack
local ledNumber = 29                    -- размерность буфера
local leds = Ledbar.new(ledNumber)      -- объект управления лентой

local function changeColor(col)
  for i=0, ledNumber-1 do               -- индексы 0..28
    leds:set(i, unpack(col))            -- записываем RGB
  end
end

changeColor({0,1,0})                    -- включаем зелёный
```

Что происходит. Функция `changeColor` принимает таблицу `{r, g, b}`, а `unpack` раскладывает её в три аргумента `leds:set`. Цикл `for i = 0, ledNumber - 1` обходит все 29 индексов. Один вызов `changeColor({0, 1, 0})` — и вся линейка зелёная.

Индекс вне диапазона не вызывает ошибку

`leds:set(29, ...)` на линейке из 29 элементов молча игнорируется: светодиод не загорится, а программа продолжит работать. Если какой-то светодиод «не светится», первым делом проверьте индекс.

Исследование: какой индекс где

lua/05_leds/leds_addressing_research.lua

```lua
local leds = Ledbar.new(29)

-- Функция для очистки (выключения всех)
local function clear()
    for i=0, 28 do leds:set(i, 0,0,0) end
end

clear()
-- Включаем индекс 0
leds:set(0, 1, 0, 0)
-- Если загорелся не тот, который вы ожидали — поменяйте 0 на другой
↪  индекс от 0 до 28
```

Задание 4.1: Карта светодиодов

Запустите программу: загорится светодиод с индексом `0`. Посмотрите на дрон (в симуляторе — камерой «**Дрон**»): какой это угол? Замените `0` на `1`, `2`, `3` и заполните таблицу. Если есть модуль матрицы — продолжите до `28`.

Индекс	Расположение
0	.....
1	.....
2	.....
3	.....
4 ...28	.....

Это упражнение учит главному принципу инженера: если документации недостаточно — проведи эксперимент.

4.5. Время: почему не sleep

Чтобы цвет менялся, нужно время. Первое, что приходит в голову, — остановить программу на секунду функцией `sleep`:

lua/05_leds/leds_sleep.lua

```lua
local ledNumber = 29
local leds = Ledbar.new(ledNumber)

local function fill(r, g, b)
  for i = 0, ledNumber - 1 do leds:set(i, r, g, b) end
end

fill(1, 0, 0)      -- красный
sleep(1)           -- вся программа стоит одну секунду: ни событий, ни
↪  таймеров
fill(0, 1, 0)      -- зелёный
sleep(1)
fill(0, 0, 0)      -- выключить

function callback(event)
end
```

Запустите: красный, через секунду зелёный, ещё через секунду линейка гаснет. Работает. Но пока `sleep` «спит», программа не делает ничего другого: не принимает события автопилота, не обслуживает другие таймеры, не читает датчики. На земле это терпимо, в полёте — нет: дрон не узнает, что долетел до точки, и не отреагирует на разряд батареи.

Без sleep

В программах для дрона `sleep` не используется. Пауза — это таймер: «через t секунд сделай вот это», а программа тем временем свободна.

4.6. Таймеры

API таймеров

- `Timer.callLater(delay, fn)` — вызвать `fn` один раз через `delay` секунд.
- `t = Timer.new(period, fn)` — создать периодический таймер: `fn` вызывается каждые `period` секунд, пока таймер запущен.
- `t:start()` и `t:stop()` — запустить и остановить периодический таймер.

Время всегда в секундах: `0.5` — полсекунды.

Отложенное действие

lua/05_leds/leds_timers_delayed.lua

```lua
local ledNumber = 29
local leds = Ledbar.new(ledNumber)
Timer.callLater(3, function ()
  for i=0, ledNumber-1 do leds:set(i, 0, 0, 1) end  -- включить синий
end)
function callback(event)
end
```

Программа заканчивается сразу, но через 3 секунды автопилот вызывает переданную функцию, и линейка загорается синим. Обратите внимание: в `callLater` передаётся *функция*, а не результат её вызова.

Периодическое действие: мигание

lua/05_leds/leds_timers_blink.lua

```lua
local ledNumber = 29
local leds = Ledbar.new(ledNumber)
local on = false
function updateBlink()
  local r,g,b = (on and 1 or 0), 0, 0
  for i=0, ledNumber-1 do leds:set(i, r, g, b) end
  on = not on
end
blinkTimer = Timer.new(0.5, updateBlink)  -- 0.5 секунды
blinkTimer:start()

function callback(event)
end
```

Что происходит. Переменная `on` хранит состояние мигалки. Каждые 0.5 с функция `updateBlink` сначала выставляет цвет по текущему состоянию (`on and 1 or 0` — единица, если `on` истинно), а затем меняет состояние на противоположное (`on = not on`) — на следующем тике цвет будет другим. Первый тик линейку гасит, второй зажигает красным, и так по кругу. Таймер `blinkTimer` объявлен глобально — так он живёт до конца программы, и его можно остановить из любой функции.

Один тик — один шаг

Функция таймера должна делать один короткий шаг анимации и возвращаться. Никаких циклов ожидания внутри: следующий шаг таймер вызовет сам.

Задачи

1. Измените период мигания на 0.2 с и цвет на синий.
2. Через 2 с после запуска включите зелёный, через 4 с — выключите (два `callLater`).
3. «Бегущий огонёк»: таймер с периодом 0.1 с зажигает только один светодиод, индекс которого увеличивается на 1 и после 28 возвращается к 0.

4.7. События автопилота

Таймер — это событие «прошло время». Автопилот умеет сообщать и о других событиях: моторы запущены, взлёт завершён, точка достигнута, дрон сел, батарея разряжена, удар. Для них есть одна функция:

Обработчик событий

```
function callback(event)
  if event == Ev.COPTER_LANDED then
    -- что сделать после посадки
  end
end
```

Мы её только определяем. Вызывает её автопилот — каждый раз, когда что-то произошло, — и передаёт в `event` код события. Коды хранятся в таблице `Ev`; сравнивать нужно именно с `Ev.ИМЯ`, а не с числом.

Событие	Когда приходит
`Ev.ENGINES_STARTED`	моторы запущены после предстартовой подготовки
`Ev.TAKEOFF_COMPLETE`	взлёт завершён, дрон висит на высоте взлёта
`Ev.POINT_REACHED`	достигнута точка, заданная `ap.goToLocalPoint`
`Ev.COPTER_LANDED`	дрон коснулся земли после посадки
`Ev.LOW_VOLTAGE1, Ev.LOW_VOLTAGE2`	батарея разряжена: первый порог — предупреждение, второй — пора садиться
`Ev.SHOCK`	удар или столкновение

Реакция на посадку

lua/05_leds/leds_events_landed.lua

```lua
local ledNumber = 29
local leds = Ledbar.new(ledNumber)

-- пока программа работает, линейка горит белым
for i = 0, ledNumber - 1 do leds:set(i, 1, 1, 1) end

function callback(event)
  if event == Ev.COPTER_LANDED then                          -- дрон сел
    for i = 0, ledNumber - 1 do leds:set(i, 0, 0, 0) end     -- погасить
    ↪ линейку
  end
end
```

Линейка горит белым всё время работы и гаснет, когда автопилот сообщит о посадке. В симуляторе проверить это можно так: добавьте в начало программы взлёт и посадку из главы 5 — или запустите готовый пример `leds_mission_stages.lua` из следующего раздела.

Реакция на разряд батареи

lua/05_leds/leds_events_low_voltage.lua

```lua
local ledNumber = 29
local leds = Ledbar.new(ledNumber)
local on = false

local function fill(r, g, b)
  for i = 0, ledNumber - 1 do leds:set(i, r, g, b) end
end

-- штатная работа: мигаем зелёным два раза в секунду
blinkTimer = Timer.new(0.5, function()
  on = not on
  fill(0, on and 1 or 0, 0)
end)
blinkTimer:start()

function callback(event)
  if event == Ev.LOW_VOLTAGE2 then                           -- батарея
  ↪ разряжена
    blinkTimer:stop()                                        -- мигание
    ↪ больше не нужно
    Timer.callLater(1, function() fill(1, 0, 0) end)         -- через
    ↪ секунду — ровный красный
  end
```

```
end
```

Что происходит. В штатном режиме таймер мигает зелёным. Событие `LOW_VOLTAGE2` останавливает таймер и через секунду заливает линейку красным. Пауза нужна, потому что уже запланированный тик таймера может сработать ещё раз сразу после `stop()` — так устроен автопилот Пионера.

Короткий вариант того же приёма, без таймера мигания. Останавливать здесь нечего, поэтому секундная пауза не обязательна — это просто отложенная реакция на событие:

lua/05_leds/leds_timers_callback_delayed.lua

```
local ledNumber = 29
local leds = Ledbar.new(ledNumber)
function callback(event)
  if (event == Ev.LOW_VOLTAGE2) then
    Timer.callLater(1, function ()
      for i=0, ledNumber-1 do leds:set(i, 1, 0, 0) end
    end)
  end
end
```

callback короткий и быстрый

Внутри `callback` — только смена состояния, цвет и команда автопилоту. Никаких `sleep` и длинных циклов: пока обработчик работает, следующие события ждут.

4.8. Три способа организовать программу

4.8.1. 1. Последовательность с sleep

Действие, пауза, действие. Просто, но блокирует программу: в полёте недопустимо. Годится только для проверки на земле.

4.8.2. 2. Таймеры

Каждая анимация — свой таймер, каждый тик — один шаг. Таймеры независимы: мигание навигационных огней и перелив матрицы работают одновременно, а программа остаётся свободной для событий.

4.8.3. 3. Конечный автомат

Когда поведение зависит от того, «что уже произошло», одной переменной-флага мало. Конечный автомат — это:

- множество состояний (`"RED"`, `"YELLOW"`, `"GREEN"`);
- события (тик таймера, `Ev.POINT_REACHED`, тумблер пульта);
- правила переходов: «в состоянии A пришло событие X → перейти в B и сделать Y».

Светофор — самый маленький автомат:

lua/05_leds/leds_traffic_light.lua

```lua
local ledNumber = 29
local leds = Ledbar.new(ledNumber)

local function fill(color)
  for i = 0, ledNumber - 1 do leds:set(i, table.unpack(color)) end
end

-- состояния автомата и цвет каждого из них
local colors = {RED = {1, 0, 0}, YELLOW = {1, 1, 0}, GREEN = {0, 1, 0}}
-- таблица переходов: из какого состояния в какое
local nextState = {RED = "YELLOW", YELLOW = "GREEN", GREEN = "RED"}

state = "RED"                                -- текущее состояние
 ↪  (глобальная переменная)
fill(colors[state])

trafficTimer = Timer.new(2, function()       -- событие "прошло 2 секунды"
  state = nextState[state]                   -- переход в следующее
   ↪  состояние
  fill(colors[state])                        -- действие нового состояния
end)
trafficTimer:start()

function callback(event)
end
```

Что происходит. Таблица `nextState` — это и есть правила переходов, `colors` — действие в каждом состоянии. Таймер приносит событие «прошло 2 секунды», обработчик делает переход и действие. Добавить четвёртое состояние — значит добавить по строке в две таблицы, не трогая логику.

4.9. Примеры комплексных сценариев

Перелив цвета и аварийная индикация

Этот скрипт — официальный пример Геоскан, слегка причёсанный:

lua/05_leds/leds_random_color_example.lua

```lua
-- https://learnxinyminutes.com/docs/ru-ru/lua-ru/ ссылка для быстрого
↪  ознакомления с основами языка LUA

-- Скрипт реализует плавный перелив цвета всей линейки и красный цвет
↪  при разряде батареи

-- Упрощение вызова функции распаковки таблиц из модуля table
local unpack = table.unpack
-- Количество светодиодов: 4 на плате + 25 на модуле матрицы
local ledNumber = 29
-- Создание порта управления светодиодами
local leds = Ledbar.new(ledNumber)

-- Функция, изменяющая цвет всех RGB светодиодов линейки
local function changeColor(col)
    for i=0, ledNumber - 1, 1 do
        leds:set(i, unpack(col))
    end
end

-- Функция, реализующая выключение таймера и изменение цвета светодиодов
↪  на красный
local function emergency()
    -- Остановка таймера
    timerRandomLED:stop()
    -- Изменение цвета светодиодов (через секунду - т.к. после остановки
    ↪  таймера timerRandomLED
    -- его функция выполнится еще раз)
    Timer.callLater(1, function () changeColor({1, 0, 0}) end)
end

-- Функция обработки событий, автоматически вызывается автопилотом
function callback(event)
    -- Вызов функции emergency() при низком напряжении на аккумуляторе
    if (event == Ev.LOW_VOLTAGE2) then
        emergency()
    end
end

color = {0, 0, 0}
step = 0.05
index = 1
```

```
-- Создание таймера: каждые 0.05 с он чуть сдвигает одну компоненту
↪  цвета (перелив)
timerRandomLED = Timer.new(0.05, function ()
    color[index] = color[index] + step
    if color[index] > 0.95 or color[index] < 0.05 then
        if index < 3 then
            index = index + 1
        else
            index = 1
            step = -step
        end
    end
    changeColor(color)
end)
-- Запуск созданного таймера
timerRandomLED:start()
```

Таймер каждые 0.05 с чуть сдвигает одну из компонент цвета — получается плавный перелив. Событие `LOW_VOLTAGE2` через функцию `emergency` останавливает таймер и включает красный. Два механизма — таймер и событие — работают вместе, не мешая друг другу.

Индикация этапов полёта

А это — мост к следующей главе. Программа взлетает, летит к точке и садится, а цвет линейки показывает этап:

lua/05_leds/leds_mission_stages.lua

```
local ledNumber = 29
local leds = Ledbar.new(ledNumber)

local function fill(r, g, b)
  for i = 0, ledNumber - 1 do leds:set(i, r, g, b) end
end

fill(1, 1, 1)                                    -- белый: подготовка
ap.push(Ev.MCE_PREFLIGHT)
Timer.callLater(1, function() ap.push(Ev.MCE_TAKEOFF) end)

function callback(event)
  if event == Ev.TAKEOFF_COMPLETE then
    fill(0, 1, 0)                                -- зелёный: летим к
    ↪  точке
    ap.goToLocalPoint(0, 0.5, 0.8)
  end
  if event == Ev.POINT_REACHED then
```

```
    fill(0, 0, 1)                                   -- синий: садимся
    ap.push(Ev.MCE_LANDING)
  end
  if event == Ev.COPTER_LANDED then
    fill(0, 0, 0)                                   -- сели: погасить
  end
end
```

Белый — подготовка, зелёный — полёт к точке, синий — посадка, выключено — дрон на земле. Команды `ap.push` и `ap.goToLocalPoint` разобраны в главе 5; здесь важно другое: каждая смена цвета стоит рядом с переходом на новый этап, и по дрону видно, где программа. Переменной `state` в этой программе нет: этапы идут строго друг за другом, и каждое событие (`TAKEOFF_COMPLETE`, `POINT_REACHED`, `COPTER_LANDED`) приходит ровно один раз, поэтому само событие однозначно говорит, какой этап начался. Когда одно и то же событие может прийти на разных этапах, понадобится явная переменная состояния — так устроены миссии в разделе 5.7.

4.10. Проверьте себя

1. Сколько светодиодов адресует `Ledbar.new(29)` и какие индексы допустимы?
   (a) 29, индексы 1..29
   (b) 29, индексы 0..28
   (c) 28, индексы 0..28
   Ответ: (b).
2. Как включить зелёный на всей линейке функцией `changeColor`?
   (a) `changeColor({0,1,0})`
   (b) `changeColor({0,255,0})`
   (c) `changeColor(0,1,0)`
   Ответ: (a) — таблица с компонентами 0..1.
3. Как включить синий через 3 секунды, не останавливая программу?
   (a) `sleep(3); fill(0,0,1)`
   (b) `Timer.callLater(3, function() fill(0,0,1) end)`
   (c) `Timer.new(3, fill)`
   Ответ: (b).

4. Какой период задать для мигания раз в полсекунды? `Timer.new(___, fn)`
   (a) 0.5
   (b) 500
   (c) 50
   Ответ: (a) — секунды.
5. Как остановить таймер `t`?
   (a) `t:stop()`
   (b) `stop(t)`
   (c) `t.cancel()`
   Ответ: (a).
6. Как называется функция, которую вызывает автопилот при событии?
   (a) `onEvent(e)`
   (b) `callback(event)`
   (c) `handler(ev)`
   Ответ: (b).
7. Какое событие соответствует посадке?
   (a) `Ev.POINT_REACHED`
   (b) `Ev.COPTER_LANDED`
   (c) `Ev.LOW_VOLTAGE2`
   Ответ: (b).
8. Что хранит переменная `state` в светофоре?
   (a) Номер светодиода
   (b) Текущее состояние автомата
   (c) Период таймера
   Ответ: (b).

4.11. Практика

Решения всех заданий лежат в `examples/lua-tasks/05_leds/` — загляните туда только после своей попытки.

Базовые операции (без таймеров)

Задание 4.2: Чередование по чётности

Светодиоды с чётными индексами — красные `{1,0,0}`, с нечётными — синие `{0,0,1}`. Узор задаётся один раз. Подсказка: `i % 2 == 0`.

Задание 4.3: Полоса загрузки

Светодиоды от `0` до `k` включены зелёным, остальные выключены; `k` задаётся в программе.

Таймеры и автоматы

Задание 4.4: Бегущий огонёк

В каждый момент горит ровно один зелёный светодиод; каждые 0.1 с он сдвигается на следующий индекс, после 28 — снова 0.

Задание 4.5: Светофор с четырьмя состояниями

Добавьте в светофор состояние `"RED_YELLOW"` между красным и зелёным, как у настоящего светофора. Сколько строк пришлось изменить?

Задание 4.6: Плавная пульсация

Яркость красного канала растёт от 0 до 1 шагами по 0.05 каждые 0.05 с, затем убывает, и так по кругу.

Задание 4.7: Стробоскоп

Белые вспышки: 0.05 с включено, 0.2 с выключено. Подсказка: два состояния с разной длительностью — это `callLater` из обработчика, а не один периодический таймер.

Задание 4.8: Полицейская мигалка

Чётные светодиоды красные, нечётные синие; каждые 0.3 с цвета меняются местами.

Задание 4.9: Цветовой шум

Каждые 0.2 с каждому светодиоду назначаются случайные компоненты `math.random()`.

Задание 4.10: Сигнал SOS

Три короткие вспышки (0.2 с), три длинные (0.6 с), три короткие; пауза между вспышками 0.2 с, между буквами 0.6 с. Подсказка: таблица длительностей и автомат с индексом текущей вспышки.

Задание 4.11: Двоичный счётчик секунд

Раз в секунду на светодиодах `0..5` отображается число секунд с запуска в двоичном виде.

Реакция на события

Задание 4.12: Посадка гасит линейку

Линейка горит зелёным; по `Ev.COPTER_LANDED` выключается. Проверьте в симуляторе, добавив взлёт и посадку из примера `leds_mission_stages.lua`.

Задание 4.13: Авария

По `Ev.LOW_VOLTAGE2` все таймеры останавливаются, линейка горит ровным красным; по `Ev.SHOCK` — мигает красным с периодом 0.2 с.

Итоги главы

- Линейка — 29 светодиодов с индексами `0..28`; цвет — три числа `0..1`.
- `Ledbar.new(29)` один раз, `leds:set(i, r, g, b)` сколько угодно; обход — `for i = 0, 28`.
- Паузы — только таймерами: `callLater` для одного действия, `Timer.new` для повторяющихся.
- События автопилота приходят в `callback(event)`; сравнивать с `Ev.ИМЯ`.
- Сложное поведение — конечный автомат: переменная состояния плюс таблица переходов.

5. Автономные миссии

Цели главы

- Знать жизненный цикл миссии: предстарт → взлёт → точки → посадка.
- Отдавать команды автопилоту (`ap.push`, `ap.goToLocalPoint`) и дожидаться их выполнения по событиям.
- Понимать систему координат сцены и задавать точки в метрах.
- Летать по фигурам: прямая, многоугольник, окружность — точками и таймером.
- Строить миссию как конечный автомат с аварийными реакциями и запуском с пульта.

5.1. Что такое миссия

Автономная миссия — заранее запрограммированная последовательность действий, которую дрон выполняет без участия пилота. Главное, что нужно понять в этой главе: автопилот Пионера — **не исполнитель команд по порядку**, а машина событий. Программа *отдаёт команду* («взлетай») и заканчивается. Когда автопилот выполнит команду, он *сообщит об этом событием* (`TAKEOFF_COMPLETE`) — и только тогда программа отдаёт следующую. Поэтому программа миссии — это не список команд, а набор ответов на вопрос «что делать, когда произошло X». Ответы живут в `callback(event)`, и мы уже умеем его писать.

5.2. Толковый словарь

Автопилот (`ap`) модуль команд полёта: предстарт, взлёт, посадка, полёт к точке, курс.

Команда (`ap.push(Ev.MCE_*)`) просьба автопилоту сменить режим. Выполняется не мгновенно.

Событие (`Ev.*` в `callback`) сообщение автопилота о том, что команда выполнена или что-то произошло.

Локальная система координат оси X, Y на плоскости и Z вверх, начало — точка старта сцены, единицы — метры.

Курс (yaw) угол поворота носа дрона вокруг вертикальной оси, в радианах.

Счётчик шагов переменная с номером текущей точки маршрута: по ней программа понимает, какую точку только что достигла.

Failsafe аварийная реакция: что делать при ударе, разряде батареи, потере связи.

RC-каналы (`Sensors.rc`) значения стиков и тумблеров пульта, от -1 до 1.

5.3. Жизненный цикл миссии

Этапы любой миссии

1. **Инициализация** — переменные, таймеры, светодиоды. Дрон на земле.
2. **Предстарт** — `ap.push(Ev.MCE_PREFLIGHT)`: проверка датчиков, запуск моторов. Ответ — `Ev.ENGINES_STARTED`.
3. **Взлёт** — `ap.push(Ev.MCE_TAKEOFF)`: подъём на высоту взлёта (параметр автопилота `Flight_com_takeoffAlt`, около метра). Ответ — `Ev.TAKEOFF_COMPLETE`.
4. **Маршрут** — `ap.goToLocalPoint(x, y, z)` столько раз, сколько точек. Ответ на каждую — `Ev.POINT_REACHED`.
5. **Возврат** — ещё одна точка над стартом: `ap.goToLocalPoint(0, 0, z)`. Отдельной команды нет.
6. **Посадка** — `ap.push(Ev.MCE_LANDING)`: снижение и касание. Ответ — `Ev.COPTER_LANDED`, моторы выключаются сами.

Команды и события парами

Команда	Что делает автопилот	Событие по завершении
`ap.push(Ev.MCE_PREFLIGHT)`	проверяет датчики, запускает моторы	`Ev.ENGINES_STARTED`
`ap.push(Ev.MCE_TAKEOFF)`	взлетает на высоту взлёта	`Ev.TAKEOFF_COMPLETE`
`ap.goToLocalPoint(x, y, z)`	летит в точку, метры	`Ev.POINT_REACHED`
`ap.updateYaw(rad)`	поворачивает нос, радианы	`---`
`ap.push(Ev.MCE_LANDING)`	снижается и садится	`Ev.COPTER_LANDED`
`ap.push(Ev.ENGINES_DISARM)`	немедленно выключает моторы	`---`

ENGINES_DISARM в воздухе — это падение

Команда выключает моторы мгновенно, где бы дрон ни был. Её место — после `COPTER_LANDED` или в аварийной реакции на удар, когда дрон уже неуправляем. Для штатного завершения — только `MCE_LANDING`.

Две формы ожидания

Между предстартом и взлётом официальные примеры Геоскан выдерживают паузу таймером: `Timer.callLater(1, function() ap.push(Ev.MCE_TAKEOFF) end)`. Можно и строже — взлетать по событию `ENGINES_STARTED`. Оба варианта правильны; в книге чаще первый, потому что он короче. Но для точек маршрута ожидание по времени не годится: сколько секунд займёт перелёт, заранее неизвестно, поэтому следующая точка — только по `POINT_REACHED`.

callback обязателен

Даже если программе не нужны события, объявите пустую `function callback(event) end`: так требует автопилот Пионера — без неё скрипт на борту может остановиться после первой команды миссии. Симулятор такую программу выполнит, но предупредит об отсутствии `callback` в уведомлениях.

5.4. Координаты: где точка

Точки задаются в **локальной системе координат** в **метрах**:

- ось **X** направлена вперёд (туда, куда смотрит нос дрона на старте), ось **Y** — влево, ось **Z** — вверх;
- начало координат $(0, 0, 0)$ — стартовая площадка. Это фиксированная точка сцены, а не место последнего взлёта: `ap.goToLocalPoint(0, 0, 1)` всегда возвращает дрон в точку над стартом;
- на настоящем Пионере координаты даёт система позиционирования (ультразвуковая LPS или оптическая); в симуляторе — сцена.



Рис. 2: Камера «**Земля**» — вид сверху, оси как на компасе в правом верхнем углу. `Ctrl`+щелчок по земле или объекту показывает координаты точки, которые можно подставить в `ap.goToLocalPoint`.

Z — это высота цели, а не поверхности

`Ctrl`+щелчок показывает высоту поверхности, по которой щёлкнули. Лететь нужно выше: над крышей высотой 13 м задавайте $z = 14$. Над ровной площадкой удобная учебная высота — от 0.7 до 1.5 м.

Курс

Курс (`yaw`) — поворот носа вокруг вертикальной оси. Автопилот принимает его в **радианах**: $\pi/2 \approx 1.57$ — четверть оборота влево. Переводить градусы удобно функцией `math.rad(deg)`. Команда `ap.updateYaw(rad)` меняет курс, не меняя точку; её часто вызывают перед `goToLocalPoint`, чтобы дрон летел носом вперёд.

Задание 5.1: Разведка площадки

Включите камеру «**Земля**» и `Ctrl`+щёлкните по четырём точкам вокруг старта. Запишите их координаты — они понадобятся для маршрута в следующем разделе.

5.5. Ступени полёта

Каждая программа ниже — на одну ступень сложнее предыдущей. Запускайте каждую, смотрите камерой «**Дрон**» и читайте терминал.

Ступень 1. Только моторы

lua/06_missions/missions_engines_arm.lua

```lua
ap.push(Ev.MCE_PREFLIGHT)                    -- безопасная предстартовая
 ↪  подготовка
Timer.callLater(1, function()                -- отложить действие на 1
 ↪  секунду
  ap.push(Ev.ENGINES_ARM)                    -- команда запуска двигателей
   ↪  (без взлёта)
end)

function callback(event)
end
```

Предстарт, через секунду — команда `ap.push(Ev.ENGINES_ARM)`: запуск моторов без взлёта (она есть в справочнике, приложение А). Так проверяют моторы на настоящем дроне со снятыми пропеллерами. Пустой `callback` обязателен.

Ступень 2. Взлёт и посадка

lua/06_missions/mission_takeoff_landing.lua

```lua
ap.push(Ev.MCE_PREFLIGHT)                    -- предстарт
Timer.callLater(1, function()                -- через 1 секунду
  ap.push(Ev.MCE_TAKEOFF)                    -- инициировать взлёт
end)

function callback(event)                      -- обработчик системных
 ↪  событий
  if event == Ev.TAKEOFF_COMPLETE then       -- взлёт завершён
    ap.push(Ev.MCE_LANDING)                  -- выполнить посадку
  end
  if event == Ev.COPTER_LANDED then          -- посадка завершена
    ap.push(Ev.ENGINES_DISARM)               -- выключить двигатели
  end
end
```

Что происходит. Предстарт → через секунду взлёт → по событию `TAKEOFF_COMPLETE` посадка → по `COPTER_LANDED` команда `ENGINES_DISARM`. После посадки автопилот и сам выключает моторы, так что эта строка — страховка; зато она показывает единственное безопасное место для `ENGINES_DISARM` в штатной миссии. Четыре строки в `callback` — это вся миссия.

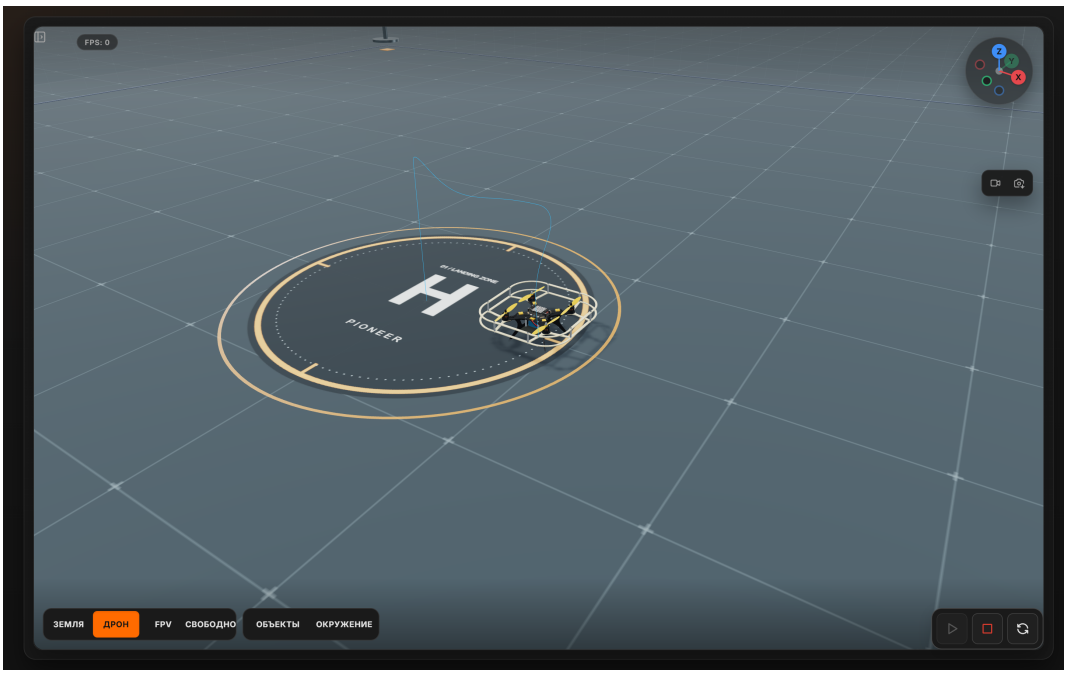


Рис. 3: Камера «**Дрон**» следует за дроном; голубая линия — след полёта.

Ступень 3. Полёт по прямой

lua/06_missions/missions_straight_flight.lua

```lua
ap.push(Ev.MCE_PREFLIGHT)                  -- предстарт
Timer.callLater(1, function()              -- через 1 секунду
  ap.push(Ev.MCE_TAKEOFF)                  -- взлёт
end)

local unpack = table.unpack                -- сокращение доступа к
↪  распаковке
local target = {1.0, 0.0, 0.8}             -- целевая точка (x,y,z)

function callback(event)                    -- обработчик событий
  if event == Ev.TAKEOFF_COMPLETE then      -- после взлёта
    ap.goToLocalPoint(unpack(target))       -- команда полёта к точке
  end
  if event == Ev.POINT_REACHED then         -- точка достигнута
    ap.push(Ev.MCE_LANDING)                 -- выполнить посадку
  end
end
```

После взлёта — точка $(1.0, 0.0, 0.8)$: метр вперёд на высоте 0.8 м. Посадка — по `POINT_REACHED`, то есть когда дрон действительно там.

Посадка раньше точки

Частая ошибка — отправить точку и в том же обработчике, через таймер на пару секунд, — посадку. Дрон летит со скоростью около 0.5 м/с: за две секунды он не долетит, и посадка отменит перелёт. Следующая команда — по следующему событию.

Ступень 4. Несколько точек и счётчик шагов

Каждая точка заканчивается одним и тем же событием `POINT_REACHED`. Чтобы знать, *какая* точка достигнута, нужен счётчик:

lua/06_missions/missions_points_flight.lua

```lua
local unpack = table.unpack                 -- сокращение для распаковки
local points = {                            -- набор точек (маршрут)
  {0.0, 0.5, 0.8}, {0.5, 0.5, 0.8}, {0.5, 0.0, 0.8}, {0.0, 0.0, 0.8}
}
local curr = 1                              -- номер следующей точки
↪  маршрута

local function nextPoint()                  -- функция перехода к
↪  следующей точке
  if curr <= #points then                   -- если есть непройденные
  ↪  точки
    ap.goToLocalPoint(unpack(points[curr])) -- команда полёта
    curr = curr + 1                         -- увеличить индекс
  else
    ap.push(Ev.MCE_LANDING)                 -- иначе посадка
  end
end

ap.push(Ev.MCE_PREFLIGHT)                   -- предстарт
Timer.callLater(1, function()               -- через 1 секунду
  ap.push(Ev.MCE_TAKEOFF)                   -- взлёт
end)

function callback(event)                     -- обработчик событий
  if event == Ev.TAKEOFF_COMPLETE then nextPoint() end -- старт маршрута
  if event == Ev.POINT_REACHED then nextPoint() end    -- шаг по
  ↪  достижению точки
end
```

Что происходит. Маршрут — таблица точек, `curr` — номер следующей. Функция `nextPoint` отправляет точку с номером `curr` и увеличивает счётчик; когда точки кончились — садится. Её вызывают два события:

TAKEOFF_COMPLETE (первая точка) и POINT_REACHED (все остальные). Это основной шаблон маршрута в курсе; так же устроены официальные примеры Геоскан.

Ступень 5. Фигуры из точек

Треугольник — три вершины на окружности радиуса r под углами $0°$, $120°$, $240°$: $x = r\cos a$, $y = r\sin a$.

lua/06_missions/missions_triangle_flight.lua

```lua
local unpack = table.unpack                    -- распаковка таблицы
local r = 0.6                                  -- радиус описанной окружности
local z = 0.9                                  -- высота полёта
local angles = {0, 120, 240}                   -- углы вершин (градусы)
local points = {}                              -- массив координат вершин
for i=1, #angles do                            -- вычисление координат
  local a = angles[i] * math.pi / 180          -- радианы
  points[i] = {r*math.cos(a), r*math.sin(a), z} -- (x,y,z) вершины
end
local curr = 1                                 -- текущая вершина
local function nextPoint()                     -- переход к следующей
  if curr <= #points then                      -- если есть вершины
    ap.goToLocalPoint(unpack(points[curr])) -- команда полёта
    curr = curr + 1                            -- увеличить индекс
  else
    ap.push(Ev.MCE_LANDING)                    -- завершение — посадка
  end
end
ap.push(Ev.MCE_PREFLIGHT)                      -- предстарт
Timer.callLater(1, function() ap.push(Ev.MCE_TAKEOFF) end) -- взлёт
function callback(event)                        -- обработчик событий
  if event == Ev.TAKEOFF_COMPLETE then nextPoint() end     -- первый шаг
  if event == Ev.POINT_REACHED then nextPoint() end        -- шаг по
  ↪ событию
end
```

Квадрат со стороной a и центром в начале координат — вершины $(\pm a/2, \pm a/2, z)$:

lua/06_missions/missions_square_flight.lua

```lua
local unpack = table.unpack                    -- распаковка таблиц
local a = 1.0                                  -- длина стороны квадрата
local z = 0.9                                  -- высота полёта
local h = a/2                                  -- половина стороны
↪ (координаты ±h)
```

```
local points = {                              -- вершины против часовой
 ↪  стрелки (вид сверху)
  {-h, -h, z}, {h, -h, z}, {h,  h, z}, {-h,  h, z}
}
local curr = 1                                -- индекс текущей вершины
local function nextPoint()                    -- переход к следующей вершине
  if curr <= #points then                     -- если есть вершины
    ap.goToLocalPoint(unpack(points[curr])) -- полёт к вершине
    curr = curr + 1                           -- увеличить индекс
  else
    ap.push(Ev.MCE_LANDING)                   -- завершение — посадка
  end
end
ap.push(Ev.MCE_PREFLIGHT)                     -- предстарт
Timer.callLater(1, function() ap.push(Ev.MCE_TAKEOFF) end) -- взлёт
function callback(event)                       -- обработчик событий
  if event == Ev.TAKEOFF_COMPLETE then nextPoint() end     -- первый шаг
  if event == Ev.POINT_REACHED then nextPoint() end        -- шаг по
   ↪  достижению
end
```

Время перелёта

У `ap.goToLocalPoint` есть четвёртый аргумент — время перелёта в секундах: `ap.goToLocalPoint(1, 0, 0.8, 6)` долетит за 6 с. Без него скорость берётся из параметров автопилота.

Задание 5.2: Свой маршрут

Подставьте в `missions_points_flight.lua` четыре точки, записанные в задании «Разведка площадки», и добавьте пятую — возврат в точку над стартом $(0, 0, 0.8)$ перед посадкой.

5.6. Окружность: точки или таймер

Окружность радиуса r в плоскости XY задаётся параметрически:

$$x(\theta) = r\cos\theta, \qquad y(\theta) = r\sin\theta, \qquad z = \text{const},$$

где θ — угол в радианах. Если удобнее думать в градусах, переводите: $\theta = \text{deg} \cdot \pi/180$ или `math.rad(deg)`.

Есть два способа пролететь окружность.

Способ 1. Много точек

Разбить окружность на 12 точек через 30° и пройти их счётчиком шагов, как любой маршрут. Дрон будет притормаживать в каждой вершине — получится двенадцатиугольник. Так устроен пример с индикацией в разделе 5.7.

Способ 2. Таймер двигает цель

Каждые 0.1 с сдвигать угол на несколько градусов и отправлять новую точку. Дрон не успевает остановиться — цель всё время впереди — и летит плавно:

lua/06_missions/missions_circle_timer.lua

```lua
local r = 1.0                                -- радиус, м
local height = 1.0                           -- высота, м
local angle = 0                              -- текущий угол, градусы
local state = "CIRCLE"                       -- CIRCLE -> RETURN -> LANDING

-- дрон действительно в точке (x, y, z)? допуск 0.2 м
local function near(x, y, z)
  local px, py, pz = Sensors.lpsPosition()
  return math.sqrt((px - x)^2 + (py - y)^2 + (pz - z)^2) < 0.2
end

circleTimer = Timer.new(0.1, function()      -- тик каждые 0.1 с
  if state ~= "CIRCLE" then return end       -- лишний тик после stop()
  ↪  ничего не делает
  if angle >= 360 then                       -- круг пройден
    circleTimer:stop()
    state = "RETURN"
    ap.goToLocalPoint(0, 0, height)          -- возвращаемся в центр
    return
  end
  local rad = math.rad(angle)                -- градусы -> радианы
  ap.goToLocalPoint(r * math.cos(rad), r * math.sin(rad), height)
  angle = angle + 5                          -- шаг 5 град.: полный круг за
  ↪  72 тика
end)

ap.push(Ev.MCE_PREFLIGHT)
Timer.callLater(1, function() ap.push(Ev.MCE_TAKEOFF) end)

function callback(event)
  if event == Ev.TAKEOFF_COMPLETE then
    circleTimer:start()
  end
  -- POINT_REACHED последней точки круга мог прийти уже после перехода в
  ↪  RETURN,
  -- поэтому садимся, только если дрон действительно над центром
```

```
    if event == Ev.POINT_REACHED and state == "RETURN" and near(0, 0,
    ↪  height) then
      state = "LANDING"
      ap.push(Ev.MCE_LANDING)                -- центр достигнут — садимся
    end
  end
```

Что происходит.

- Таймер стартует только после `TAKEOFF_COMPLETE`: отправлять точки до взлёта нельзя.
- Пока дрон кружит, события `POINT_REACHED` приходят постоянно — и игнорируются: переменная `state` равна `"CIRCLE"`. Здесь впервые одно и то же событие означает разное в зависимости от этапа, поэтому программа хранит состояние и проверяет его вместе с событием: `event == Ev.POINT_REACHED and state == "RETURN"`. Подробно этот приём разобран в разделе 5.7.
- После 360° таймер останавливает сам себя, переводит автомат в `"RETURN"` и отправляет центр. Следующий `POINT_REACHED` в состоянии `"RETURN"` означает «центр достигнут», и только тогда посадка.
- Функция `near` сверяет с этим событием реальное положение дрона (`Sensors.lpsPosition()`, допуск 0.2 м). Событие о последней точке круга могло прийти чуть позже перехода в `"RETURN"`; без проверки программа приняла бы его за «центр» и посадила дрон на окружности. Правило: если событие могло опоздать, проверяй не только событие, но и то, что оно описывает.
- Первая строка таймера, `if state ~= "CIRCLE" then return end`, — защита: тик, уже запланированный в момент `stop()`, может сработать ещё раз (глава 4), и без проверки он отправил бы лишнюю точку.

Курс по касательной

Чтобы дрон летел «носом вперёд», курс должен быть касательным к окружности: $\psi = \theta + 90°$. В том же тике, что и точка, обновляется курс:

lua/06_missions/missions_circle_yaw_timer.lua

```lua
local r = 0.3                                  -- радиус окружности, м
local height = 0.7                             -- высота полёта, м
local step = 3                                 -- шаг угла за тик, градусы
local angle = 0                                -- текущий угол точки, градусы
local state = "CIRCLE"                         -- CIRCLE -> RETURN -> LANDING

-- дрон действительно в точке (x, y, z)? допуск 0.2 м
local function near(x, y, z)
  local px, py, pz = Sensors.lpsPosition()
  return math.sqrt((px - x)^2 + (py - y)^2 + (pz - z)^2) < 0.2
end

circleTimer = Timer.new(0.1, function()        -- каждые 0.1 с: новая точка и
↪  новый курс
  if state ~= "CIRCLE" then return end         -- лишний тик после stop() не
  ↪  должен затереть возврат в центр
  angle = (angle + step) % 360
  local rad = angle * math.pi / 180            -- градусы -> радианы
  ap.updateYaw(rad + math.pi / 2)              -- нос по касательной к
  ↪  окружности
  ap.goToLocalPoint(r * math.cos(rad), r * math.sin(rad), height)
end)

ap.push(Ev.MCE_PREFLIGHT)                      -- предстарт
Timer.callLater(2, function() ap.push(Ev.MCE_TAKEOFF) end) -- взлёт
↪  через 2 с

function callback(event)
  if event == Ev.TAKEOFF_COMPLETE then
    circleTimer:start()                        -- начинаем кружить
    Timer.callLater(12, function()             -- один круг: 360 / 3 шага *
    ↪  0.1 с = 12 с
      circleTimer:stop()
      state = "RETURN"
      ap.goToLocalPoint(0, 0, height)          -- возвращаемся в центр
    end)
  end
  -- садимся только над центром: запоздалый POINT_REACHED точки круга не
  ↪  в счёт
  if event == Ev.POINT_REACHED and state == "RETURN" and near(0, 0,
  ↪  height) then
    state = "LANDING"
    ap.push(Ev.MCE_LANDING)
  end
end
```

Здесь круг ограничен временем: $360°/3° \cdot 0.1$ с $= 12$ с, после чего таймер останавливается, дрон возвращается в центр и садится по событию. Длительность круга — как раз «пауза по смыслу»: сам автопилот о конце круга не

сообщает. Та же проверка `state ~= "CIRCLE"` в начале таймера не даёт запоздалому тику затереть точку возврата в центр.

Задание 5.3: Восьмёрка

Две окружности с центрами $(\pm r, 0)$, проходимые в противоположных направлениях. Подсказка: два состояния `"LEFT"` и `"RIGHT"`, шаг угла разного знака, переход — когда угол прошёл $360°$.

5.7. Архитектура миссии: конечный автомат

Три принципа

1. **События, а не время.** Следующая команда — по событию о завершении предыдущей. Таймер — только для пауз, которые нужны по смыслу.
2. **Ничего не блокировать.** Ни `sleep`, ни циклов ожидания: `callback` должен возвращаться мгновенно.
3. **Failsafe.** Удар (`Ev.SHOCK`) — выключить моторы; разряд (`Ev.LOW_VOLTAGE2`) — прекратить маршрут и сесть.

Когда у миссии несколько этапов, одной переменной-счётчика мало: одно и то же событие `POINT_REACHED` в состоянии «выхожу на окружность» и в состоянии «возвращаюсь в центр» означает разное. Поэтому программа хранит **состояние** и проверяет его вместе с событием. В светофоре (глава 4) событие было одно — тик таймера, — и правила переходов умещались в таблицу `nextState[state]`. В миссии событий несколько, и переход зависит от пары «состояние + событие»; поэтому правила записываются условиями вида `event == Ev.X and state == "Y"`.

Окружность по точкам с индикацией

lua/06_missions/missions_lesson12_circle_points.lua

```lua
-- Полёт по окружности из 12 точек с индикацией этапов
local ledNumber = 29
local leds = Ledbar.new(ledNumber)
local unpack = table.unpack

local function changeColor(color)
```

```
  for i = 0, ledNumber - 1 do leds:set(i, unpack(color)) end
end

local colors = {
  red = {1, 0, 0}, white = {1, 1, 1}, green = {0, 1, 0},
  blue = {0, 0, 1}, off = {0, 0, 0}
}

-- точки окружности: 12 штук через 30 град.
local r = 0.5
local height = 0.8
local points = {}
for i = 1, 12 do
  local a = math.rad(i * 30)
  points[i] = {r * math.cos(a), r * math.sin(a), height}
end

local curr_point = 1
local state = "PREPARE"                    -- PREPARE -> FLIGHT ->
↪  LANDING; при ударе EMERGENCY

local function nextPoint()
  if curr_point <= #points then
    ap.goToLocalPoint(unpack(points[curr_point]))
    curr_point = curr_point + 1
  else
    state = "LANDING"
    changeColor(colors.blue)               -- синий: садимся
    ap.push(Ev.MCE_LANDING)
  end
end

function callback(event)
  if event == Ev.TAKEOFF_COMPLETE and state == "PREPARE" then
    state = "FLIGHT"
    changeColor(colors.green)              -- зелёный: маршрут
    nextPoint()
  end
  if event == Ev.POINT_REACHED and state == "FLIGHT" then
    nextPoint()
  end
  if event == Ev.COPTER_LANDED and state == "LANDING" then
    changeColor(colors.off)                -- сели: погасить
  end
  if event == Ev.SHOCK then
    state = "EMERGENCY"                    -- маршрут больше не
    ↪  продолжаем
    changeColor(colors.red)                -- авария: красный и
    ↪  выключить моторы
    ap.push(Ev.ENGINES_DISARM)
  end
```

```
end

changeColor(colors.red)                          -- красный: программа
↪  загружена
Timer.callLater(2, function()
  changeColor(colors.white)                      -- белый: предстарт
  ap.push(Ev.MCE_PREFLIGHT)
end)
Timer.callLater(4, function() ap.push(Ev.MCE_TAKEOFF) end)
```

Что происходит. Состояния: PREPARE → FLIGHT → LANDING. Цвет линейки показывает этап: красный — программа загружена, белый — предстарт, зелёный — маршрут, синий — посадка, выключено — на земле. POINT_REACHED обрабатывается только в состоянии FLIGHT, поэтому случайное событие на посадке ничего не сломает; так же TAKEOFF_COMPLETE принимается только в PREPARE, а COPTER_LANDED гасит линейку только после штатной посадки. SHOCK в любом состоянии — красный, ENGINES_DISARM и переход в EMERGENCY: из него маршрут уже не продолжается, и красный цвет остаётся.

Непрерывная окружность с курсом

lua/06_missions/missions_lesson14_circle.lua

```
-- Непрерывный полёт по окружности: точка и курс обновляются таймером
local ledNumber = 29
local leds = Ledbar.new(ledNumber)
local unpack = table.unpack

local function changeColor(color)
  for i = 0, ledNumber - 1 do leds:set(i, unpack(color)) end
end

local colors = {
  red = {1, 0, 0}, white = {1, 1, 1}, yellow = {1, 1, 0},
  purple = {1, 0, 1}, blue = {0, 0, 1}, off = {0, 0, 0}
}

local r = 0.5                                    -- радиус, м
local height = 0.8                               -- высота, м
local step = 3                                   -- градусов за тик
local angle = 0
local state = "PREPARE"                          -- PREPARE -> TO_CIRCLE ->
↪  CIRCLE -> RETURN -> LANDING; при ударе EMERGENCY
```

```
local function near(x, y, z)                -- дрон действительно в
↪  точке? допуск 0.2 м
  local px, py, pz = Sensors.lpsPosition()
  return math.sqrt((px - x)^2 + (py - y)^2 + (pz - z)^2) < 0.2
end

circleTimer = Timer.new(0.1, function()
  if state ~= "CIRCLE" then return end      -- лишний тик после stop() не
  ↪  должен затереть возврат в центр
  angle = (angle + step) % 360
  local rad = math.rad(angle)
  ap.updateYaw(rad + math.pi / 2)           -- нос по касательной к
  ↪  окружности
  ap.goToLocalPoint(r * math.cos(rad), r * math.sin(rad), height)
end)

function callback(event)
  if event == Ev.TAKEOFF_COMPLETE and state == "PREPARE" then
    state = "TO_CIRCLE"
    changeColor(colors.yellow)              -- жёлтый: выходим на
    ↪  окружность
    ap.goToLocalPoint(r, 0, height)
  elseif event == Ev.POINT_REACHED and state == "TO_CIRCLE" then
    state = "CIRCLE"
    changeColor(colors.purple)              -- фиолетовый: кружим
    circleTimer:start()
    Timer.callLater(12, function()          -- один круг: 360 / 3 шага *
    ↪  0.1 с
      if state ~= "CIRCLE" then return end -- после аварии круг уже
      ↪  прерван
      circleTimer:stop()
      state = "RETURN"
      changeColor(colors.blue)              -- синий: возвращаемся в
      ↪  центр
      ap.goToLocalPoint(0, 0, height)
    end)
  elseif event == Ev.POINT_REACHED and state == "RETURN" and near(0, 0,
  ↪  height) then
    state = "LANDING"
    ap.push(Ev.MCE_LANDING)
  elseif event == Ev.COPTER_LANDED and state == "LANDING" then
    changeColor(colors.off)
  elseif event == Ev.SHOCK then
    state = "EMERGENCY"                     -- никакие таймеры и события
    ↪  больше не ведут маршрут
    circleTimer:stop()
    changeColor(colors.red)                 -- авария: красный и
    ↪  выключить моторы
    ap.push(Ev.ENGINES_DISARM)
  end
end
```

```
changeColor(colors.white)                    -- белый: предстарт
ap.push(Ev.MCE_PREFLIGHT)
Timer.callLater(2, function() ap.push(Ev.MCE_TAKEOFF) end)
```

Что происходит. Пять состояний: `PREPARE` → `TO_CIRCLE` (жёлтый, выход в точку $(r, 0)$) → `CIRCLE` (фиолетовый, таймер двигает цель и курс 12 с) → `RETURN` (синий, точка над центром) → `LANDING`. Цепочка `elseif` в `callback` — это и есть таблица переходов: каждая строка — «событие и состояние → действие». `SHOCK` переводит автомат в `EMERGENCY`; таймер окружности и отложенный конец круга (`Timer.callLater(12, ...)`) первым делом проверяют, что автомат всё ещё в `CIRCLE`, поэтому после аварии они не отправят точку и не перекрасят линейку. Посадка в `RETURN` — только когда `near(0, 0, height)` подтверждает, что дрон над центром (раздел 5.6).

Аварийные реакции

lua/06_missions/missions_failsafe.lua

```
-- Патруль между двумя точками с аварийными реакциями
local ledNumber = 29
local leds = Ledbar.new(ledNumber)
local points = {{1, 0, 1}, {-1, 0, 1}}
local curr = 1
local state = "TAKEOFF"                      -- TAKEOFF -> PATROL ->
↪  LANDING; при ударе EMERGENCY
local lowBattery = false                     -- разряд пришёл, пока дрон
↪  взлетал

local function fill(r, g, b)
  for i = 0, ledNumber - 1 do leds:set(i, r, g, b) end
end

local function nextPoint()
  ap.goToLocalPoint(table.unpack(points[curr]))
  curr = curr % #points + 1                  -- 1 -> 2 -> 1 -> ...
end

local function land()
  state = "LANDING"
  fill(1, 0, 0)
  ap.push(Ev.MCE_LANDING)                    -- садимся там, где летим
end

function callback(event)
```

```
  if event == Ev.SHOCK then                    -- удар в любом состоянии:
   ↪  немедленно выключить моторы
    state = "EMERGENCY"
    fill(1, 0, 0)
    ap.push(Ev.ENGINES_DISARM)
  elseif event == Ev.TAKEOFF_COMPLETE and state == "TAKEOFF" then
    if lowBattery then
      land()                                   -- батарея села ещё на
       ↪  взлёте: маршрут не начинаем
    else
      state = "PATROL"
      fill(0, 1, 0)                            -- зелёный: патрулируем
      nextPoint()
    end
  elseif event == Ev.POINT_REACHED and state == "PATROL" then
    nextPoint()
  elseif event == Ev.LOW_VOLTAGE2 then
    if state == "PATROL" then
      land()                                   -- батарея садится:
       ↪  прекращаем маршрут
    elseif state == "TAKEOFF" then
      lowBattery = true                        -- посадку во время взлёта
       ↪  автопилот не примет: сядем после него
      fill(1, 0, 0)
    end                                        -- в LANDING и EMERGENCY
     ↪  повторное событие ничего не меняет
  end
end

ap.push(Ev.MCE_PREFLIGHT)
Timer.callLater(1, function() ap.push(Ev.MCE_TAKEOFF) end)
```

Что происходит. Дрон патрулирует между двумя точками бесконечно. Состояния: `TAKEOFF` → `PATROL` → `LANDING`, при ударе — `EMERGENCY`. Одно событие получает разную реакцию в зависимости от состояния:

- `SHOCK` — в любом состоянии сразу выключение моторов: аварию не откладывают.
- `LOW_VOLTAGE2` в `PATROL` — посадка на месте. Во время взлёта автопилот команду посадки не примет, поэтому в `TAKEOFF` программа только запоминает разряд (`lowBattery`) и садится сразу после `TAKEOFF_COMPLETE`, не начиная маршрут. В `LANDING` и `EMERGENCY` повторное событие (напряжение может «прыгать» около порога) ничего не меняет.
- `POINT_REACHED` ведёт маршрут только в `PATROL`.

В симуляторе эти события можно вызвать с панели «**Полёт**».

Запуск с пульта

На настоящем дроне миссию удобно запускать тумблером: программа загружена, дрон стоит, пилот отходит и переводит тумблер. `Sensors.rc()` возвращает восемь каналов (значения от −1 до 1), тумблер SwA — восьмой:

lua/06_missions/missions_rc_initiation.lua

```lua
local unpack = table.unpack
local points = {{0, 2, 1}, {0, 1, 1}, {0, 2, 1}, {0, 1, 1}} -- маршрут
local curr_point = 1
local started = false                        -- миссия уже запущена?

local function nextPoint()
  if curr_point <= #points then
    ap.goToLocalPoint(unpack(points[curr_point]))
    curr_point = curr_point + 1
  else
    ap.push(Ev.MCE_LANDING)                  -- точки кончились — посадка
  end
end

function callback(event)
  if event == Ev.TAKEOFF_COMPLETE then Timer.callLater(1, nextPoint) end
  if event == Ev.POINT_REACHED then Timer.callLater(1, nextPoint) end
end

-- опрашиваем тумблер SwA (8-й канал) раз в секунду, пока миссия не
↪  запущена
startTimer = Timer.new(1, function()
  local rc_chans = table.pack(Sensors.rc()) -- 8 значений каналов в
  ↪  таблицу
  if rc_chans[8] > 0 and not started then    -- тумблер SwA переключён:
  ↪  значение канала больше нуля
    started = true
    startTimer:stop()                        -- больше не опрашиваем
    ap.push(Ev.MCE_PREFLIGHT)
    Timer.callLater(2, function() ap.push(Ev.MCE_TAKEOFF) end)
  end
end)
startTimer:start()
```

Что происходит. Таймер раз в секунду читает каналы. Как только восьмой канал положителен — флаг `started`, таймер останавливается (иначе он отдавал бы предстарт каждую секунду), предстарт, через 2 с взлёт. Дальше маршрут ведёт уже знакомый счётчик шагов; `Timer.callLater(1, nextPoint)` — секундное зависание в каждой точке: следующая команда по-прежнему отда-

ётся только после события, просто с паузой. Настройка пульта и соответствие тумблеров каналам — в приложении C; в симуляторе пульт эмулирует панель «**Пульт**».

Светодиоды для отладки

В примерах этого раздела с индикацией цвет меняется вместе с состоянием. Это дешёвый и надёжный журнал: на полигоне без терминала по цвету видно, на каком этапе программа зависла. Приучите себя подсвечивать состояния с самого начала.

5.8. Частые ошибки

Перед запуском симулятор проверяет программу и объясняет найденное в уведомлениях (колокольчик в заголовке редактора), а во время полёта останавливает её, если команды противоречат друг другу. Вот что он чаще всего находит:

- **Команды подряд без ожидания.** `ap.push(Ev.MCE_PREFLIGHT)` и сразу `ap.push(Ev.MCE_TAKEOFF)`. Взлёт — после `ENGINES_STARTED` или через таймер, точка — после `TAKEOFF_COMPLETE`, посадка — после `POINT_REACHED`.
- **Посадка раньше точки.** Таймер короче перелёта — дрон сядет, не долетев. Садиться по событию.
- **Нет `callback`.** На борту такая программа останавливается после первой команды; симулятор предупреждает.
- **Все точки по одному событию.** Без счётчика шагов вторая точка затирает первую. Шаблон — `nextPoint` с `curr`.
- **Точка «в земле».** `Ctrl`+щелчок показывает высоту поверхности; z цели должен быть выше.
- **Градусы вместо радиан.** `ap.updateYaw(90)` — это 90 радиан, то есть 14 оборотов. Нужно `math.rad(90)`.
- **Таймер не остановлен.** Таймер окружности продолжает слать точки после команды посадки — дрон не сядет. `t:stop()` перед сменой этапа.
- **Функция вызвана вместо передачи.** `Timer.callLater(1, nextPoint())` выполняет `nextPoint` сразу; нужно `Timer.callLater(1, nextPoint)`.

Где смотреть

Терминал (вкладка «**Вывод**») показывает `print` и сообщения автопилота о каждом переходе: предстарт, взлёт, точка достигнута. Панель «**Полёт**» показывает текущее состояние автопилота и координаты. Если дрон «завис» — сначала туда.

5.9. Проверьте себя

1. Как начать миссию? `ap.push(___)`
   - **(a)** `Ev.MCE_PREFLIGHT`
   - **(b)** `Ev.START`
   - **(c)** `Ev.MCE_TAKEOFF`

   Ответ: (a) — предстарт всегда первый.
2. Какое событие сообщает, что взлёт завершён?
   - **(a)** `Ev.ENGINES_STARTED`
   - **(b)** `Ev.TAKEOFF_COMPLETE`
   - **(c)** `Ev.POINT_REACHED`

   Ответ: (b).
3. Как задать курс $90°$?
   - **(a)** `ap.updateYaw(90)`
   - **(b)** `ap.updateYaw(math.rad(90))`
   - **(c)** `ap.updateYaw(90 / 180)`

   Ответ: (b) — радианы.
4. Когда отправлять вторую точку маршрута?
   - **(a)** Через 4 с после первой
   - **(b)** По `POINT_REACHED` первой
   - **(c)** Сразу вслед за первой

   Ответ: (b).
5. Как безопасно выполнить действие через 2 секунды?
   - **(a)** `sleep(2); action()`
   - **(b)** `Timer.callLater(2, action)`
   - **(c)** `Timer.new(2, action)`

   Ответ: (b).
6. Как организовать непрерывное обновление точки окружности?

(a) `while true do ap.goToLocalPoint(...) end`

(b) `Timer.new(0.1, function() ap.goToLocalPoint(...) end)`

(c) `repeat ... until false`

Ответ: (b).

7. Что делать по `Ev.SHOCK`?

(a) Продолжить маршрут

(b) `ap.push(Ev.MCE_LANDING)`

(c) Остановить таймеры и `ap.push(Ev.ENGINES_DISARM)`

Ответ: (c).

8. Где в `rc_chans = table.pack(Sensors.rc())` тумблер SwA?

(a) `rc_chans[1]`

(b) `rc_chans[8]`

(c) `rc_chans["SwA"]`

Ответ: (b).

5.10. Практика

Решения — в `examples/lua-tasks/07_advanced_missions/`. Во всех заданиях: предстарт и взлёт как в примерах, светодиодная индикация этапов, реакции на `SHOCK` и `LOW_VOLTAGE2`.

Базовые миссии

Задание 5.4: Многоугольник из N точек

Сгенерируйте N вершин окружности радиуса 0.5 м под углами $360^\circ \cdot i/N$, пройдите их счётчиком шагов, сядьте. Проверьте для $N = 3, 6, 12$.

Задание 5.5: Два круга

Пролетите квадрат со стороной 1 м дважды подряд и только потом садитесь. Подсказка: счётчик кругов.

Задание 5.6: Спираль

На каждом тике таймера радиус растёт на `dr`, высота — на `dz`. При $z \geq z_{max}$ остановить таймер, вернуться в точку над стартом и сесть.

Задание 5.7: Восьмёрка

Две окружности с противоположным направлением обхода и общим центром между ними; завершение — в центре и посадка.

Задание 5.8: Старт и останов тумблером

Тумблер вверх — предстарт и взлёт, окружность; тумблер вниз — прекратить, вернуться в точку над стартом, сесть.

Задание 5.9: Лоитер

Удержание малой окружности $r = 0.2$ м неограниченно долго с остановкой по тумблеру или `LOW_VOLTAGE2`.

Продвинутые

Задание 5.10: Радиус с пульта

Седьмой канал (`rc_chans[7]`, от -1 до 1) задаёт радиус окружности от 0.2 до 1.0 м: $r = 0.6 + 0.4 \cdot \mathrm{ch7}$.

Задание 5.11: Ограничение скорости поворота

Курс меняется не быстрее `maxYawRate` рад/с: на тике таймера приращение ограничивается `maxYawRate * dt`.

Задание 5.12: Повтор после посадки

По `COPTER_LANDED` через 3 с миссия начинается заново; после третьего круга — выключение моторов.

Задание 5.13: Лепесток

Траектория $r = r_0\,(1 + 0.2 \sin k\theta)$ при $k \geq 3$, параметр — угол таймера.

Задание 5.14: Точность навигации

Пройдите 8 точек окружности и в терминал печатайте интервалы времени между событиями `POINT_REACHED` (`time()` возвращает секунды). Насколько они одинаковы?

Задание 5.15: Демонстрация

Старт с пульта, окружность около двух минут, индикация этапов, посадка, выключение моторов после `COPTER_LANDED`.

Итоги главы

- Миссия — цепочка «команда → событие → команда»; цепочку ведёт `callback`.
- Предстарт, взлёт, точки, посадка; счётчик шагов для маршрута; таймер для плавных траекторий.
- Координаты в метрах от старта, Z вверх; курс в радианах.
- Сложная миссия — конечный автомат: состояние проверяется вместе с событием.
- Failsafe обязателен: `SHOCK` — моторы выключить, `LOW_VOLTAGE2` — сесть.

Часть II.

Программирование на Python

6. Основы Python

Цели главы

- Понять, чем Python отличается от Lua, и запустить первую программу.
- Работать с переменными, типами, арифметикой и логикой.
- Управлять ходом программы: ветвления `if`/`elif`/`else`, циклы `for` и `while`.
- Писать функции — строительный материал для программ полёта в следующей главе.

Как работать с главой

Примеры запускайте в симуляторе, выбрав язык «**Python**»: вывод `print` появится в терминале, как и для Lua. Структура главы повторяет главу про Lua, поэтому сравнивайте конструкции двух языков — это лучший способ запомнить оба.

6.1. Что такое Python и зачем он нужен?

Python — это высокоуровневый язык программирования, который был создан Гвидо ван Россумом и впервые выпущен в 1991 году. Назван он не в честь змеи, а в честь популярного британского комедийного шоу «Летающий цирк Монти Пайтона».

Почему Python?

- **Простота:** Синтаксис Python похож на обычный английский язык.
- **Универсальность:** Используется в веб-разработке, науке, AI и робототехнике.
- **Мощь:** Огромное количество библиотек (OpenCV, NumPy, TensorFlow) для решения сложных задач.

Для программирования квадрокоптеров Pioneer мы выбрали Python, так как он позволяет легко взаимодействовать с оборудованием и обрабатывать видеопоток с камеры.

Сравнение Python и Lua

Мы уже знакомы с языком Lua, который используется для написания скриптов автопилота. Давайте посмотрим, чем Python отличается от Lua:

Характеристика	Lua	Python
Назначение	Встраиваемый скриптовый язык (для игр, контроллеров)	Универсальный язык общего назначения
Синтаксис	Использует `end` для блоков кода	Использует **отступы** для блоков кода
Индексация	С 1 (первый элемент — 1)	С 0 (первый элемент — 0)
Библиотеки	Ограниченный набор	Огромная экосистема (PyPI)
Скорость	Очень быстрый	Медленнее, но мощнее

Таблица 1: Сравнение Lua и Python

Важное отличие

Главное, что нужно запомнить: в Python блоки кода (циклы, функции, условия) выделяются **отступами** (обычно 4 пробела), а не ключевыми словами `do ... end`.

Первая программа

Традиционно изучение любого языка начинается с программы, которая выводит приветствие на экран.

python/01_python_basics/01_hello_world.py

```python
# Первая программа на Python
print("Hello, World!")
```

Здесь функция `print()` выводит текст, заключенный в кавычки, в консоль.

Задание 6.1: Привет, мир!

Напишите программу, которая выводит ваше имя и фамилию на экран.

6.2. Переменные и типы данных

Переменная — это именованная область памяти, в которой хранится значение. В Python не нужно заранее объявлять тип переменной, он определяется автоматически при присваивании.

Основные типы данных

- `int` (integer) — целые числа (например, `10`, `-5`).
- `float` — числа с плавающей точкой (например, `3.14`, `1.5`).
- `str` (string) — строки текста (например, `"Pioneer"`).
- `bool` (boolean) — логические значения (`True`, `False`).

python/01_python_basics/02_variables.py

```python
# Переменные и типы данных
name = "Pioneer"  # Строка (str)
altitude = 1.5    # Число с плавающей точкой (float)
battery = 100     # Целое число (int)
is_flying = False # Логический тип (bool)

print("Дрон:", name)
print("Высота:", altitude)
print("Тип переменной altitude:", type(altitude))
```

Функция `type()` позволяет узнать тип переменной.

Задание 6.2: Личная карточка

Создайте переменные для хранения вашего имени, возраста и любимого предмета. Выведите их на экран одной строкой.

Задачи для самостоятельного решения

1. Создайте переменную `speed` и присвойте ей значение 5. Выведите её на экран.
2. Создайте переменную `height` и присвойте ей значение 1.5. Выведите её.

3. Создайте переменную `name` со значением `"Pioneer"`. Выведите её.
4. Создайте переменную `score` = 10. Затем измените её значение на 20. Выведите `score`.
5. Создайте переменную `x` = 100. Затем присвойте ей строку `"Сто"`. Выведите тип переменной до и после изменения.
6. Даны `a = 5` и `b = 7`. Создайте переменную `c`, равную их сумме. Выведите `c`.
7. Даны `part1 = "Geo"` и `part2 = "skan"`. Создайте `full`, сложив их.
8. Создайте строку `s = "Fly"`. Выведите её 3 раза подряд (используйте умножение).
9. Какого типа переменная `pi = 3.1415`? Выведите результат функции `type(pi)`.
10. Присвойте переменным `x`, `y`, `z` значения 1, 2, 3 в одной строке.
11. Даны `a = 10`, `b = 20`. Поменяйте их значения местами без использования третьей переменной.
12. Создайте переменную `is_flying` со значением `True`. Выведите её.
13. Дано целое число `n = 5`. Преобразуйте его в дробное и сохраните в `f`.
14. Дана строка `s_num = "123"`. Преобразуйте её в число `num` и прибавьте 1.
15. Даны `mass = 10` и `accel = 9.8`. Вычислите `force = mass * accel`.

6.3. Арифметические операторы

Python поддерживает все стандартные математические операции.

- `+` Сложение
- `-` Вычитание
- `*` Умножение
- `/` Деление (всегда возвращает `float`)
- `//` Целочисленное деление (отбрасывает дробную часть)
- `%` Остаток от деления
- `**` Возведение в степень

python/01_python_basics/03_arithmetic.py

```
# Арифметические операции
x = 10
y = 3
```

```
print("Сложение:", x + y)
print("Деление:", x / y)
print("Целочисленное деление:", x // y) # 3
print("Остаток от деления:", x % y)     # 1
print("Возведение в степень:", x ** y)  # 1000
```

Обратите внимание на приоритет операций: умножение и деление выполняются раньше сложения и вычитания, как в математике. Скобки `()` используются для изменения порядка действий.

Задание 6.3: Калькулятор

Напишите программу, которая считает площадь прямоугольника со сторонами 5 и 8 метров.

Задачи для самостоятельного решения

1. Сложите числа 15 и 38. Выведите результат.
2. Вычтите из 100 число 45. Выведите результат.
3. Умножьте 7 на 8.
4. Разделите 50 на 4. Какой тип данных получился?
5. Сколько раз число 4 помещается в 50 целиком? Используйте `//`.
6. Чему равен остаток от деления 20 на 3? Используйте `%`.
7. Возведите 2 в степень 10 (2 в 10-й степени).
8. Вычислите `2 + 2 * 2`. Затем `(2 + 2) * 2`. Сравните результаты.
9. Сторона квадрата `a = 5`. Найдите периметр `P = 4 * a`.
10. Даны стороны `a = 10`, `b = 20`. Найдите площадь `S = a * b`.
11. Сколько секунд в одном часе? (60 секунд * 60 минут).
12. Даны числа 10, 20, 30. Найдите их среднее арифметическое.
13. Дано число 12345. Найдите его последнюю цифру, используя оператор `% 10`.
14. Дано четное число `n = 10`. Получите следующее четное число (+2).
15. Умножьте -5 на -5. Проверьте знак результата.

6.4. Логические операции

Логические операторы позволяют комбинировать условия. Результатом всегда является `True` (Истина) или `False` (Ложь).

Операторы сравнения

- `==` Равно
- `!=` Не равно
- `>` Больше
- `<` Меньше
- `>=` Больше или равно
- `<=` Меньше или равно

Логические связки

- `and` (И) — истина, только если оба условия истинны.
- `or` (ИЛИ) — истина, если хотя бы одно условие истинно.
- `not` (НЕ) — инвертирует значение (делает истину ложью и наоборот).

python/01_python_basics/04_logic.py

```python
# Логические операции
voltage = 3.8
is_connected = True

# Проверка условий
is_low_battery = voltage < 3.5
ready_to_fly = is_connected and not is_low_battery

print("Низкий заряд?", is_low_battery)
print("Готов к полету?", ready_to_fly)
```

Задание 6.4: Проверка безопасности

Создайте переменную `wind_speed = 7`. Напишите логическое выражение, которое возвращает `True`, если скорость ветра меньше 5 м/с.

Задачи для самостоятельного решения

1. Проверьте, больше ли 10 числа 5.

2. Проверьте, меньше или равно 5 числу 5.
3. Равны ли числа 100 и 100?
4. Правда ли, что 10 не равно 20?
5. Проверьте условие: (5 > 0) и (5 < 10).
6. Проверьте условие: (5 > 0) или (5 > 10).
7. Инвертируйте значение `True` с помощью `not`.
8. Дано `x = 15`. Проверьте, лежит ли `x` между 10 и 20 (строго).
9. Дано `n = 4`. Проверьте, делится ли оно на 2 без остатка (`n % 2 == 0`).
10. Равны ли строки ”test” и ”Test”?
11. Дано `battery = 4.0` и `connected = True`. Разрешен ли полет? (нужно и то, и другое).
12. Дано `obstacle = False`. Есть ли опасность? (Если препятствие есть ИЛИ высота < 0.5).
13. Проверьте сложное условие: (2 + 2 == 4) и (не (3 > 5)).
14. Равны ли число 5 и строка ”5”?
15. Чему равно `bool(1)`? А `bool(0)`?

6.5. Ветвление (Условные операторы)

Условные операторы `if`, `elif` и `else` позволяют программе принимать решения в зависимости от условий.

python/01_python_basics/05_branching.py

```python
# Ветвление: Робот на перекрестке
direction = "forward"

if direction == "left":
    print("Поворачиваю налево")
elif direction == "right":
    print("Поворачиваю направо")
elif direction == "forward":
    print("Еду прямо")
else:
    print("Останавливаюсь, путь неясен")
```

Обратите внимание на отступы (4 пробела) — они обязательны в Python и обозначают блоки кода.

Задание 6.5: Светофор

Напишите программу, которая принимает цвет светофора (`"red"`, `"yellow"`, `"green"`) и выводит действие (`"Стоп"`, `"Приготовиться"`, `"Ехать"`).

Задачи для самостоятельного решения

1. Дано число `x = 10`. Если оно больше 0, выведите ”Положительное”.
2. Дано `n = 7`. Если делится на 2 без остатка — ”Четное”, иначе — ”Нечетное”.
3. Дано `age = 15`. Если `age >= 18`, выведите ”Взрослый”, иначе ”Ребенок”.
4. Дано `password = "1234"`. Если пароль совпадает с ”secret”, пишем ”Доступ разрешен”, иначе ”Отказ”.
5. Даны `a = 10`, `b = 20`. Выведите большее из них.
6. Дано `x = 0`. Используя `elif`, выведите ”Плюс”, ”Минус” или ”Ноль”.
7. Дано `color = "red"`. Если ”green” — ”Ехать”, ”yellow” — ”Ждать”, ”red” — ”Стоять”.
8. `voltage = 3.6`. Если < 3.5 — ”Посадка!”, иначе ”Полет нормальный”.
9. Делится ли число 15 на 3 и на 5 одновременно?
10. Дано `name = "Alex"`. Если длина имени > 5 букв, напишите ”Длинное”, иначе ”Короткое”.
11. Дана строка `s = ""`. Если она пустая (длина 0), напишите ”Пусто”.
12. `weight = 80`. < 50 — ”Легкий”, 50-100 — ”Средний”, > 100 — ”Тяжелый”.
13. `armed = True`, `ready = False`. Если оба `True` — ”Взлет”, иначе ”Ждать”.
14. `login = "admin"`. Если не равен ”admin”, вывести ”Ошибка”.
15. `a=5, b=2, c=8`. Найдите самое большое число из трех через `if`.

6.6. Циклы

Циклы позволяют выполнять один и тот же блок кода многократно.

Цикл for

Используется, когда известно количество повторений. Функция `range(N)` создает последовательность от 0 до N-1.

Цикл while

Выполняется, пока условие истинно. Будьте осторожны: если условие никогда не станет ложным, получится «бесконечный цикл».

python/01_python_basics/06_loops.py

```python
# Циклы
import time

# Цикл for: мигаем светодиодом 5 раз (имитация)
print("Начинаю мигать...")
for i in range(5):
    print(f"Мигание №{i+1}: ВКЛ")
    time.sleep(0.5)
    print(f"Мигание №{i+1}: ВЫКЛ")
    time.sleep(0.5)

# Цикл while: летим пока есть заряд
battery = 10
while battery > 0:
    print(f"Летим... Заряд: {battery}%")
    battery -= 2
print("Посадка!")
```

- `import time` подключает стандартный модуль `time`; `time.sleep(0.5)` — пауза на полсекунды. В отличие от Lua на борту, Python-программа работает на компьютере и автопилот не блокирует, поэтому `sleep` здесь допустим (глава 7).
- `f"Мигание №{i+1}"` — f-строка: выражение в фигурных скобках вычисляется и подставляется в текст. `battery -= 2` — то же, что `battery = battery - 2`.
- `break` — немедленно прерывает цикл.
- `continue` — переходит к следующей итерации.

Задание 6.6: Обратный отсчет

Напишите программу, которая считает от 10 до 1, а затем пишет «Поехали!». Используйте цикл `for` или `while`.

Задачи для самостоятельного решения

1. Выведите числа от 0 до 4 используя `range(5)`.
2. Выведите слово ”Привет” 3 раза.

3. Выведите числа от 1 до 5 (используйте `range(1, 6)`).
4. Выведите четные числа от 0 до 8 (`range(0, 9, 2)`).
5. Найдите сумму чисел от 1 до 100.
6. Выведите числа от 5 до 1 (`range(5, 0, -1)`).
7. Выведите таблицу умножения на 5 (от 1*5 до 10*5).
8. Дана строка ”Дрон”. Выведите каждую букву с новой строки.
9. Используя `while`, выводите `x`, пока `x < 3`. Начальное `x = 0`.
10. Ищите число 7 в диапазоне от 0 до 10. Если нашли — используйте `break`.
11. Выведите числа от 0 до 4, пропустив 2 (используйте `continue`).
12. Выведите квадраты чисел от 1 до 5.
13. Посчитайте длину строки ”Hello” с помощью цикла (без `len()`).
14. Выведите 5 строк, в каждой по одной звездочке *.
15. Выведите лесенку из звезд (1, 2, 3...).

6.7. Функции

Функция — это именованный блок кода, который можно вызывать многократно. Функции делают код чище и понятнее.
Ключевое слово `def` объявляет функцию. `return` возвращает результат.

python/01_python_basics/07_functions.py

```python
# Функции
def calculate_area(length):
    """Вычисляет площадь квадрата."""
    return length * length

side = 5
area = calculate_area(side)
print(f"Площадь квадрата со стороной {side} равна {area}")

def check_battery(voltage):
    if voltage < 3.5:
        return "Critical"
    elif voltage < 3.8:
        return "Low"
    else:
        return "Normal"

print("Статус батареи 3.6В:", check_battery(3.6))
```

Область видимости

Переменные, созданные внутри функции, не видны снаружи. Это называется локальной областью видимости.

Задание 6.7: Конвертер

Напишите функцию `celsius_to_fahrenheit(c)`, которая переводит градусы Цельсия в Фаренгейты по формуле $F = C \times 1.8 + 32$.

Задачи для самостоятельного решения

1. Напишите функцию `say_hello()`, которая печатает ”Привет!”. Вызовите её.
2. Напишите функцию `greet(name)`, которая печатает ”Привет, <name>!”.
3. Напишите функцию `add(a, b)`, которая возвращает сумму двух чисел.
4. Напишите функцию `square(x)`, которая возвращает x в квадрате.
5. Напишите функцию `my_max(a, b)`, которая возвращает большее из двух чисел.
6. Напишите функцию `is_even(n)`, которая возвращает `True`, если число четное.
7. Напишите функцию `perimeter(a, b)`, возвращающую периметр прямоугольника.
8. Напишите функцию `m_to_cm(meters)`, которая переводит метры в сантиметры.
9. Напишите функцию `check_battery(voltage)`. Если < 3.5 -> ”Low”, иначе ”OK”.
10. Напишите функцию `my_abs(x)`. Если x < 0, возвращает -x, иначе x.
11. Напишите функцию `avg(a, b)`, возвращающую среднее арифметическое.
12. Напишите функцию `log_error(msg)`, которая печатает ”ERROR: <msg>”.
13. Напишите функцию `repeat(s, n)`, возвращающую строку s, повторенную n раз.
14. Напишите функцию `flight_time(cap, cons)`, считающую время полета (емкость / расход).
15. Напишите функцию `full_name(first, last)`, возвращающую ”Фамилия Имя”.

7. Pioneer SDK: полёт из Python

Цели главы

- Запускать Python-программу в симуляторе, на настоящем дроне и с компьютера на симулятор.
- Подключаться к дрону, читать телеметрию и управлять подсветкой через `led_control`.
- Выполнять взлёт и посадку безопасно: пауза после `arm()`, ожидание после `land()`.
- Летать по точкам и фигурам, дожидаясь каждой точки через `point_reached()`.

7.1. Где запускать Python

Для управления Пионером из Python есть библиотека `pioneer_sdk`: команды взлёта, посадки, полёта в точку, светодиоды, телеметрия, камера. У Python-программы, как и у Lua, два исполнителя.

В симуляторе

Выберите в редакторе язык «**Python**». Библиотека уже «установлена»: `from pioneer_sdk import Pioneer` работает, а дрон на сцене выполняет команды. Вывод `print` — в терминале, как и для Lua.

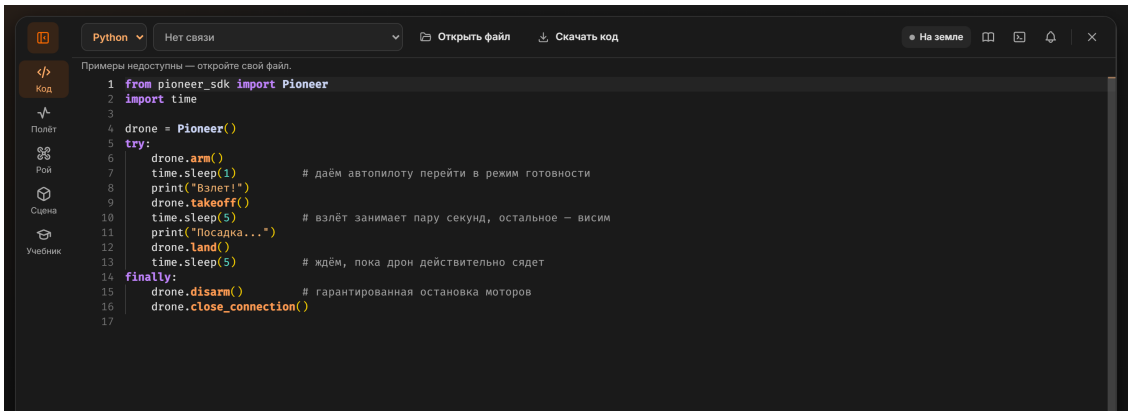


Рис. 4: Редактор в режиме Python. Переключение языка не теряет Lua-программу: у каждого языка своя вкладка.

На настоящем дроне

Python-программа работает на вашем компьютере и общается с дроном по Wi-Fi:

Терминал

```
pip install pioneer-sdk
```

Дрон создаёт сеть Wi-Fi (имя вида `Pioneer_...`); подключитесь к ней — и `Pioneer()` найдёт дрон по адресу по умолчанию.

С компьютера — на симулятор

Третий вариант: программа на вашем компьютере, а летает дрон в симуляторе. На сайте у дрона включается «**Разрешить внешние команды**», а в программе вместо адреса дрона указывается адрес сайта:

Подключение к симулятору

```
from pioneer_sdk import Pioneer
pioneer = Pioneer(ip='simulator.sakoryagin.ru',
                  mavlink_port=8001)
```

Дальше программа ничем не отличается от программы для настоящего дрона. Так удобно отлаживать то, что в браузере недоступно: сторонние библиотеки, обработку видео с камеры.

Lua ждёт событий, Python — опрашивает

В Lua следующую команду отдают из `callback` по событию. В Python программа сама периодически спрашивает автопилот: `while not pioneer.point_reached(): time.sleep(0.1)`. Смысл тот же — не отдавать следующую команду, пока предыдущая не выполнена, — меняется только форма.

7.2. Подключение и телеметрия

python/02_pioneer_sdk/08_drone_connect.py

```python
from pioneer_sdk import Pioneer
import time

# Подключение к дрону
drone = Pioneer()
print("Дрон подключен!")

try:
    # Запрашиваем состояние батареи несколько раз
    for _ in range(3):
        voltage = drone.get_battery_status()
        print(f"Напряжение: {voltage} В")
        time.sleep(1)
finally:
    # Всегда закрываем соединение
    drone.close_connection()
    print("Соединение закрыто")
```

- `Pioneer()` — объект связи с дроном. Создаётся один раз.
- `get_battery_status()` — напряжение батареи в вольтах.
- `try: ... finally: ...` — конструкция Python: сначала выполняется блок `try`, а блок `finally` выполняется *всегда* после него, даже если в `try` произошла ошибка. Поэтому `close_connection()` в `finally` закрывает связь даже при ошибке или `Ctrl+C`; иначе порт останется занят.

Задание 7.1: Диагностика

Подключитесь, прочитайте заряд и, если напряжение ниже 3.6 В, напечатайте «Low Battery!».

7.3. Светодиоды из Python

Первый видимый результат без взлёта — как и в Lua, подсветка. Метод один:

led_control

`pioneer.led_control(led_id=255, r=0, g=255, b=0)` — цвет светодиода `led_id`; `255` означает «все сразу». Компоненты — целые числа от `0` до `255` (в Lua было `0..1`). Значения вне диапазона библиотека отвергает ошибкой `ValueError`.

`(255, 0, 0)` красный	`(0, 255, 0)` зелёный	`(0, 0, 255)` синий
`(255, 255, 0)` жёлтый	`(255, 255, 255)` белый	`(0, 0, 0)` выключено

Пример связывает цвет с показаниями дальномера — датчика расстояния до поверхности под дроном:

python/03_flight_control/LED_hight_change.py

```python
from pioneer_sdk import Pioneer
import time

# Определяем временной интервал обновления данных
delta_time = 0.1

# Инициализируем переменные для цвета светодиодов (RGB)
r = 0
g = 0
b = 0

# Определяем пороговые значения высоты для смены цвета
height_1 = 0.25
height_2 = 0.5
height_3 = 0.75

#  <------- height_1 ------- height_2 ------- height_3 ------->
#   красный          зеленый           синий           белый

if __name__ == "__main__":                      # Проверяем, что скрипт
↪  запущен как основной файл
    pioneer_mini = Pioneer(logger=False)        # Создаем объект
    ↪  управления дроном, отключая логирование
    curr_time = time.time()                     # Фиксируем текущее время

    try:
        while True:                                             #
        ↪  Запускаем бесконечный цикл
            if time.time() - curr_time > delta_time:            #
            ↪  Проверяем, прошло ли заданное время (delta_time)
                tof_data = pioneer_mini.get_dist_sensor_data()  #
                ↪  Получаем данные с датчика расстояния
                print(tof_data)                                 #
                ↪  Выводим данные в консоль для мониторинга

                if tof_data is not None:                        #
                ↪  Проверяем, что данные успешно получены с датчика

                    # Определяем цвет светодиодов в зависимости от
                    ↪  высоты
                    if tof_data <= height_1:
                        r = 255  # Красный
                        g = 0
```

```
                    b = 0
                elif height_1 < tof_data <= height_2:
                    r = 0
                    g = 255  # Зеленый
                    b = 0
                elif height_2 < tof_data <= height_3:
                    r = 0
                    g = 0
                    b = 255  # Синий
                elif tof_data >= height_3:
                    r = 255
                    g = 255
                    b = 255  # Белый

                # Управляем светодиодами дрона
                pioneer_mini.led_control(led_id=255, r=r, g=g, b=b)

                # Обновляем время последнего обновления
                curr_time = time.time()
            time.sleep(0.01)                                    #
            ↪  Короткая пауза: без неё цикл грузит процессор на 100%

    except KeyboardInterrupt:
        print("Остановка программы. Выключаем светодиоды...")
        pioneer_mini.led_control(led_id=255, r=0, g=0, b=0)  # Выключаем
        ↪  светодиоды

    finally:
        pioneer_mini.close_connection()                     # Закрываем
        ↪  соединение с дроном
```

Что происходит. Бесконечный цикл `while True` раз в 0.1 с читает дальномер `get_dist_sensor_data()` и выбирает цвет по порогам высоты. В Python такой цикл допустим: программа работает на компьютере, автопилот дрона он не блокирует. Выход — `Ctrl+C`: блок `except KeyboardInterrupt` гасит подсветку, `finally` закрывает соединение. Поднимите дрон рукой (в симуляторе — перетащите) и смотрите, как меняется цвет.

Задание 7.2: Светофор батареи

Выше 4.0 В — зелёный, от 3.7 до 4.0 — жёлтый, ниже 3.7 — красный. Обновлять раз в секунду.

Задание 7.3: Первая индикация

Подключиться, включить зелёный на 2 с, выключить, корректно закрыть соединение.

7.4. Первый полёт: моторы, взлёт, посадка

Моторы

python/02_pioneer_sdk/09_01_arm_disarm.py

```python
from pioneer_sdk import Pioneer
import time

drone = Pioneer()
try:
    print("Запуск моторов...")
    drone.arm()
    time.sleep(3) # Моторы вращаются 3 секунды
finally:
    print("Остановка моторов")
    drone.disarm()
```

`arm()` запускает моторы на холостых оборотах, `disarm()` выключает. Блок `try ... finally` гарантирует выключение моторов даже при ошибке.

Взлёт и посадка

python/02_pioneer_sdk/09_02_takeoff_land.py

```python
from pioneer_sdk import Pioneer
import time

drone = Pioneer()
try:
    drone.arm()
    time.sleep(1)          # даём автопилоту перейти в режим готовности
    print("Взлет!")
    drone.takeoff()
    time.sleep(5)          # взлёт занимает пару секунд, остальное —
    ↪  висим
    print("Посадка...")
    drone.land()
    time.sleep(5)          # ждём, пока дрон действительно сядет
finally:
    drone.disarm()         # гарантированная остановка моторов
    drone.close_connection()
```

Что происходит.

- `arm()` → секунда паузы: автопилот переходит в режим готовности. Без паузы команда взлёта может быть отвергнута.
- `takeoff()` — подъём на высоту взлёта (около метра). Команда возвращается сразу, взлёт занимает пару секунд; остаток из пяти секунд дрон висит.
- `land()` — посадка. Она тоже занимает время, и `disarm()` в `finally` выполнится *сразу* после `land()`, если ничего не ждать. Поэтому после посадки — ещё `time.sleep(5)`: выключать моторы можно только на земле.

disarm в воздухе

`disarm()` выключает моторы немедленно. Вызывайте его только когда дрон сел: после `land()` подождите несколько секунд или проверьте `get_autopilot_state()`.

7.5. Полёт по точкам

Система координат

Та же, что в Lua (раздел 5): X вперёд, Y влево, Z вверх, метры от точки старта. Команда:

go_to_local_point

`pioneer.go_to_local_point(x=1.0, y=0.0, z=1.0, yaw=0)` — лететь в точку; `yaw` — курс в радианах, обязательный аргумент. Автопилот подтверждает прибытие: `pioneer.point_reached()` вернёт `True` — один раз на каждую точку.

Вперёд и назад

python/03_flight_control/10_01_forward_back.py

```python
from pioneer_sdk import Pioneer
import time

drone = Pioneer()


def fly_to(x, y, z):
    """Летим в точку и ждём, пока автопилот сообщит, что она
    ↪ достигнута."""
    drone.go_to_local_point(x=x, y=y, z=z, yaw=0)
```

```
    while not drone.point_reached():
        time.sleep(0.1)


try:
    print("Арминг и взлет...")
    drone.arm()
    time.sleep(1)            # даём автопилоту перейти в режим
    ↪ готовности
    drone.takeoff()

    h = 1.0                  # высота полёта
    fly_to(0.0, 0.0, h)      # сначала занимаем точку над стартом

    print("Вперед!")
    fly_to(1.0, 0.0, h)      # 1 метр вперёд по оси X

    print("Домой!")
    fly_to(0.0, 0.0, h)      # обратно в точку старта

    print("Посадка...")
    drone.land()
    time.sleep(5)            # ждём, пока дрон сядет
finally:
    drone.disarm()
    drone.close_connection()
```

Что происходит. Вспомогательная функция `fly_to` отправляет точку и ждёт `point_reached()`, опрашивая автопилот каждые 0.1 с. Это Python-эквивалент события `POINT_REACHED`: следующая точка — только после подтверждения предыдущей. Первая точка — над стартом: так программа дожидается конца взлёта.

Не угадывать время

`time.sleep(4)` после `go_to_local_point` — плохой способ ждать: перелёт может занять и 2, и 6 секунд. Ждите `point_reached()`.

Квадрат

python/03_flight_control/10_02_square.py

```python
from pioneer_sdk import Pioneer
import time

drone = Pioneer()


def fly_to(x, y, z):
    """Летим в точку и ждём события 'точка достигнута'."""
    drone.go_to_local_point(x=x, y=y, z=z, yaw=0)
    while not drone.point_reached():
        time.sleep(0.1)


try:
    print("Взлет...")
    drone.arm()
    time.sleep(1)
    drone.takeoff()

    h = 1.0                                  # высота полёта
    fly_to(0.0, 0.0, h)                      # точка над стартом

    # квадрат 1 x 1 м: вперёд, влево, назад, домой
    square = [(1.0, 0.0), (1.0, 1.0), (0.0, 1.0), (0.0, 0.0)]
    for x, y in square:
        print(f"Летим в ({x}, {y})")
        fly_to(x, y, h)

    drone.land()
    time.sleep(5)
finally:
    drone.disarm()
    drone.close_connection()
```

Точки — **список** пар `(x, y)`. Список в Python — упорядоченный набор значений в квадратных скобках, аналог таблицы-массива Lua, но нумерация с нуля: `square[0]` — первая вершина. Пара в круглых скобках — кортеж, неизменяемый набор значений; запись `for x, y in square` на каждом шаге раскладывает очередную пару в две переменные, как `table.unpack` в Lua. Добавить вершину — одна строка.

Окружность

python/03_flight_control/10_03_circle_math.py

```python
from pioneer_sdk import Pioneer
import time
import math

drone = Pioneer()


def fly_to(x, y, z):
    """Летим в точку и ждём события 'точка достигнута'."""
    drone.go_to_local_point(x=x, y=y, z=z, yaw=0)
    while not drone.point_reached():
        time.sleep(0.1)


try:
    print("Взлет...")
    drone.arm()
    time.sleep(1)
    drone.takeoff()

    h = 1.0              # высота полёта
    radius = 0.5         # радиус окружности, м
    points_count = 12    # чем больше точек, тем ровнее круг

    # окружность с центром в точке старта: угол от 0 град. до 360 град.
    ↪ с шагом 30 град.
    for i in range(points_count + 1):
        angle_deg = 360 / points_count * i
        angle_rad = math.radians(angle_deg)   # sin и cos принимают
        ↪ радианы
        x = radius * math.cos(angle_rad)
        y = radius * math.sin(angle_rad)
        print(f"Точка {i}: x={x:.2f}, y={y:.2f}")
        fly_to(x, y, h)

    drone.land()
    time.sleep(5)
finally:
    drone.disarm()
    drone.close_connection()
```

Двенадцать точек через $30°$: $x = R\cos\alpha$, $y = R\sin\alpha$, угол переводится в радианы функцией `math.radians`. Последняя, тринадцатая точка совпадает с первой — круг замыкается.

Задание 7.4: Восьмёрка

Соедините две окружности с центрами $(\pm 0.5, 0)$ в одну траекторию.

7.6. Управление скоростью

Точка отвечает на вопрос «куда лететь». Иногда нужно другое — «с какой скоростью лететь прямо сейчас»: держать маркер в центре кадра, плавно снижаться по дальномеру, следовать за человеком. Для этого есть команда скорости:

set_manual_speed

`pioneer.set_manual_speed(vx=0.5, vy=0, vz=0, yaw_rate=0)` — скорость в м/с по осям X, Y, Z той же системы координат, что у `go_to_local_point`, и скорость поворота `yaw_rate` в рад/с. Команда действует коротко, поэтому её повторяют в цикле каждые 0.05 с. Перестали присылать — дрон зависает на месте. Вариант `set_manual_speed_body_fixed` задаёт те же скорости относительно носа дрона.

python/03_flight_control/10_04_manual_speed.py

```python
from pioneer_sdk import Pioneer
import time

drone = Pioneer()


def move(vx, vy, vz, seconds):
    """Лететь с заданной скоростью (м/с) seconds секунд, затем зависнуть.

    Команда скорости действует коротко, поэтому её повторяют в цикле
    каждые 0.05 с — так же устроен официальный пример set_manual_speed.py.
    """
    start = time.time()
    while time.time() - start < seconds:
        drone.set_manual_speed(vx=vx, vy=vy, vz=vz, yaw_rate=0)
        time.sleep(0.05)
    drone.set_manual_speed(vx=0, vy=0, vz=0, yaw_rate=0)   # остановиться


try:
    drone.arm()
```

```
    time.sleep(1)                # даём автопилоту перейти в режим
    ↪ готовности
    drone.takeoff()
    time.sleep(5)                # взлёт и зависание

    move(0.5, 0, 0, 2)           # 2 с по оси X со скоростью 0.5 м/с
    move(0, 0.5, 0, 2)           # 2 с по оси Y
    move(0, 0, 0.3, 1)           # 1 с вверх
    time.sleep(1)

    drone.land()
    time.sleep(5)                # ждём, пока дрон сядет
finally:
    drone.disarm()
    drone.close_connection()
```

Что происходит. Функция `move` двадцать раз в секунду повторяет одну и ту же скорость, пока не пройдёт заданное время, и затем присылает нулевую скорость. Здесь ждать по времени правильно: длительность движения задаём мы сами, автопилоту «прибывать» некуда. На этой команде построены задания глав про камеру и навигацию: скорость пропорциональна ошибке (P-регулятор) и пересчитывается на каждом кадре.

7.7. Контрольные упражнения с проверкой

Для тренировки точности есть три упражнения с модулем `grader.py`: он записывает траекторию и сравнивает положение дрона с целью в контрольных точках (допуск 0.5 м по горизонтали, 0.2 м по высоте, 10° по курсу).

python/exercises/ex1_forward.py

```
from pioneer_sdk import Pioneer
from grader import FlightGrader
import time

"""
Упражнение 1 – «Полёт вперёд»: модуль должен обеспечить ровный
↪ прямолинейный проход заданной дистанции без возврата.
"""

def exercise_01():
    drone = Pioneer()
    grader = FlightGrader(drone)
```

```
    try:
        print("[System] Упражнение 1: Полет вперед")
        print("Ожидание LPS...")
        # Проверка готовности LPS (упрощенно)

        grader.start_mission()

        # 1. Взлет
        print("Взлет...")
        drone.arm()
        time.sleep(1) # Даем автопилоту перейти в режим готовности
        drone.takeoff()
        time.sleep(5)

        # 2. Установка цели (x=1.0, y=0, z=1.0)
        target_x = 1.0
        grader.set_target(target_x, 0.0, 1.0, 0.0)

        # 3. Полет вперед
        print(f"Полет к точке x={target_x}...")
        drone.go_to_local_point(x=target_x, y=0.0, z=1.0, yaw=0.0)

        # Ожидание прибытия (около 5 с)
        time.sleep(5)

        # 4. Проверка позиции
        grader.check_checkpoint("Точка Вперед")

        # 5. Посадка (Конец упражнения, без возврата)
        print("Посадка...")
        drone.land()
        time.sleep(5)  # ждём, пока дрон сядет: disarm() в finally
        ↪  выключает моторы сразу

    finally:
        grader.finish_mission()
        drone.disarm()
        drone.close_connection()

if __name__ == '__main__':
    exercise_01()
```

python/exercises/ex2_forward_turn_return.py

```
from pioneer_sdk import Pioneer
from grader import FlightGrader
import time
import math

"""
```

```
Упражнение 2 – «Полёт вперёд и назад»: модуль должен выполнить точный проход вперёд на заданное расстояние, немедленный разворот на 180 град. и возвращение на исходную точку без смещения по бокам.
"""

def exercise_02():
    drone = Pioneer()
    grader = FlightGrader(drone)

    try:
        print("[System] Упражнение 2: Вперед и Назад (с разворотом)")
        grader.start_mission()

        # 1. Взлет
        drone.arm()
        time.sleep(1) # Даем автопилоту перейти в режим готовности
        drone.takeoff()
        time.sleep(5)

        # 2. Полет вперед (x=1.0)
        dist = 1.0
        print(f"Этап 1: Вперед к x={dist}...")
        grader.set_target(dist, 0.0, 1.0, 0.0)
        drone.go_to_local_point(x=dist, y=0.0, z=1.0, yaw=0.0)
        time.sleep(5)
        grader.check_checkpoint("Точка Вперед")

        # 3. Разворот на 180 (Немедленный)
        print("Этап 2: Немедленный разворот на 180...")
        # Примечание: go_to_local_point с новым yaw. Позиция остается прежней.
        # Используем радианы для yaw, так как это стандарт.
        yaw_180 = math.radians(180)

        grader.set_target(dist, 0.0, 1.0, 180.0) # Ожидаем yaw=180 в точке x=1
        drone.go_to_local_point(x=dist, y=0.0, z=1.0, yaw=yaw_180)
        time.sleep(3)
        grader.check_checkpoint("Разворот 180")

        # 4. Возврат на старт (x=0)
        # Мы сохраняем yaw=180, то есть летим "носом" к точке старта.
        print("Этап 3: Возврат на старт...")
        grader.set_target(0.0, 0.0, 1.0, 180.0)
        drone.go_to_local_point(x=0.0, y=0.0, z=1.0, yaw=yaw_180)
        time.sleep(5)
        grader.check_checkpoint("Точка Дома (Задом)")

        drone.land()
        time.sleep(5)  # ждём, пока дрон сядет: disarm() в finally выключает моторы сразу
```

```
    finally:
        grader.finish_mission()
        drone.disarm()
        drone.close_connection()

if __name__ == '__main__':
    exercise_02()
```

Третье упражнение, `ex3_forward_turn_return.py`, отличается только строгим контролем курса на всех этапах. Файл `grader.py` должен лежать рядом с программой.

Почему здесь ждут по времени

В упражнениях после `go_to_local_point` стоит `time.sleep(5)`, а не ожидание `point_reached()`: `grader` проверяет, где дрон оказался к заданному моменту, — это часть задания на точность. В собственных программах действует правило «Не угадывать время» выше. После `land()` упражнения тоже ждут несколько секунд, чтобы `disarm()` в `finally` не выключил моторы в воздухе.

7.8. Задания к главе

Уровень 1

1. **Калькулятор полёта.** Запросить время полёта (мин) и скорость (м/с), вывести расстояние в метрах и километрах.
2. **Чек-лист.** Список из 5 пунктов предполётной проверки, вывести пронумерованным.

Уровень 2

1. **Умная батарея.** Функция `check_battery(voltage)`: выше 4.0 В — «Отличный заряд», 3.7–4.0 — «Средний», ниже 3.7 — «Посадка немедленно!». Вызвать её в скрипте подключения перед взлётом и не взлетать при низком заряде.

Уровень 3: мини-проект

Миссия «Квадрат с индикацией». Квадрат со стороной 1 м на высоте 1.5 м; перед каждой вершиной — свой цвет подсветки, после посадки — выключить. Ждать точки через `point_reached()`, а не `time.sleep`.

Итоги

- `Pioneer()` один раз; `close_connection()` и `disarm()` — в `finally`.
- `arm()` → пауза → `takeoff()` → точки через `go_to_local_point` и `point_reached()` → `land()` → подождать → `disarm()`.
- Цвета в Python — `0..255`, курс — радианы, координаты — метры.
- Точка — `go_to_local_point`; скорость — `set_manual_speed`, повторяемая в цикле.

8. Камера, компьютерное зрение и рой

Цели главы

- Получать кадры с камеры дрона и находить на них маркеры ArUco и AprilTag.
- Вычислять положение дрона относительно маркера и строить на этом навигацию.
- Читать телеметрию датчиков в полёте и записывать её.
- Распознавать человека и жесты, управлять несколькими дронами одновременно.

Нужен настоящий дрон

Симулятор отдаёт видео с виртуальной камеры и умеет кое-что из `cv2`: `detectMarkers` находит маркеры, расставленные на сцене, и возвращает их углы и номера. Но `cv2.solvePnP` в браузере всегда сообщает о неудаче, а библиотеки `mediapipe` нет совсем — `import mediapipe` завершится ошибкой. Поэтому поиск маркеров можно попробовать в симуляторе, а оценку положения, трекинг человека и всё, что ниже, — только на компьютере с реальным Пионером. Это ограничение среды, а не ошибка в программе.

8.1. Компьютерное зрение: ArUco и AprilTag

Для роботов существуют специальные визуальные метки (маркеры), которые легко распознать на изображении. Самые популярные — **ArUco** и **AprilTag**.

Что такое ArUco-маркер?

Это черно-белый квадрат с уникальным узором внутри. У каждого маркера есть свой **ID** (номер).

- **Словарь (Dictionary):** Набор допустимых маркеров. С Пионером используют маркеры 6×6: примеры этой главы берут словарь `DICT_6X6_50` (50 маркеров), карты и симулятор — `DICT_6X6_250` (250 маркеров). Противоречия нет: первые 50 маркеров `DICT_6X6_250` совпадают с `DICT_6X6_50`, поэтому маркеры с номерами 0–49 распознаются обоими словарями. Мар-

кер с номером 50 и выше нужно искать словарём `DICT_6X6_250`. Пример слежения в следующем разделе работает с маркерами 4×4 (`DICT_4X4_50`) — для него маркер печатают из этого словаря.

- **AprilTag:** Другой популярный формат, поддерживаемый той же библиотекой (например, `DICT_APRILTAG_36h11`).

Детекция маркеров

Для работы с камерой и маркерами мы используем библиотеку **OpenCV** (`cv2`).

python/aruco_examples/11_aruco_basics_example1_detect.py

```python
import cv2
import cv2.aruco as aruco
import numpy as np
from pioneer_sdk import Camera
import threading
import time

class ArucoDetectorThread:
    def __init__(self):
        self.camera = Camera()
        self.running = True
        self.latest_frame = None
        self.processed_frame = None
        self.frame_lock = threading.Lock()

        # 1. Настройка словаря ArUco (6x6, 50 маркеров - стандарт для
        ↪  Pioneer)
        self.aruco_dict =
        ↪  aruco.getPredefinedDictionary(aruco.DICT_6X6_50)

        # Обеспечение совместимости разных версий OpenCV
        if hasattr(aruco, 'DetectorParameters_create'):
            self.parameters = aruco.DetectorParameters_create()
        else:
            self.parameters = aruco.DetectorParameters()

    def detect_markers_compat(self, image):
        if hasattr(aruco, 'ArucoDetector'):
            # Новый API (OpenCV 4.7+)
            detector = aruco.ArucoDetector(self.aruco_dict,
            ↪  self.parameters)
            return detector.detectMarkers(image)
        else:
            # Старый API
```

```
            return aruco.detectMarkers(image, self.aruco_dict,
            ↪  parameters=self.parameters)

    def capture_thread(self):
        while self.running:
            frame = self.camera.get_cv_frame()
            if frame is not None:
                with self.frame_lock:
                    self.latest_frame = frame.copy()
            time.sleep(0.01)

    def processing_thread(self):
        while self.running:
            frame_to_process = None
            with self.frame_lock:
                if self.latest_frame is not None:
                    frame_to_process = self.latest_frame.copy()

            if frame_to_process is None:
                time.sleep(0.01)
                continue

            # Переводим в оттенки серого (для ускорения)
            gray = cv2.cvtColor(frame_to_process, cv2.COLOR_BGR2GRAY)

            # 3. Детекция маркеров
            corners, ids, rejected = self.detect_markers_compat(gray)

            # Если маркеры найдены
            if ids is not None:
                # Рисуем рамки вокруг маркеров
                aruco.drawDetectedMarkers(frame_to_process, corners,
                ↪  ids)
                # Выводим ID найденных маркеров в консоль (можно
                ↪  закомментировать, если спамит)
                # print(f"Вижу маркеры: {ids.flatten()}")

            with self.frame_lock:
                self.processed_frame = frame_to_process

    def run(self):
        print("Запуск детекции маркеров... Нажмите ESC для выхода.")

        t_capture = threading.Thread(target=self.capture_thread)
        t_process = threading.Thread(target=self.processing_thread)

        t_capture.daemon = True
        t_process.daemon = True

        t_capture.start()
        t_process.start()
```

```
        try:
            while True:
                display_frame = None
                with self.frame_lock:
                    if self.processed_frame is not None:
                        display_frame = self.processed_frame.copy()
                    elif self.latest_frame is not None:
                        display_frame = self.latest_frame.copy()

                if display_frame is not None:
                    cv2.imshow('Pioneer Camera', display_frame)

                if cv2.waitKey(1) == 27:
                    self.running = False
                    break

                time.sleep(0.03)

        finally:
            self.running = False
            t_capture.join(timeout=1.0)
            t_process.join(timeout=1.0)
            cv2.destroyAllWindows()

if __name__ == "__main__":
    detector = ArucoDetectorThread()
    detector.run()
```

Разбор кода

1. `camera.get_cv_frame()` получает изображение с камеры дрона. 2. `cv2.cvtColor` переводит картинку в ч/б формат (так алгоритм работает быстрее и надежнее). 3. `detectMarkers` ищет квадраты и сверяет их узор со словарем. 4. Возвращает:

- `corners`: Координаты углов найденных маркеров.
- `ids`: Список номеров (ID) найденных маркеров.

Сама программа оформлена как **класс** (`class ArucoDetectorThread`): данные хранятся в полях `self.…`, а работа разложена по методам. Методы `capture_thread` и `processing_thread` запускаются в отдельных **потоках** (`threading.Thread`) и работают одновременно с показом кадра; общий кадр защищён замком `frame_lock`, чтобы потоки не читали и не писали его в один момент. Потоки подробнее — в разделе «Групповой полет» этой главы.

Задание 8.1: Охота на маркеры

Распечатайте несколько маркеров ArUco (или откройте их на экране телефона). Напишите программу, которая пишет в консоль «Вижу маркер X!», где X — номер маркера.

Определение координат маркера

Зная размер маркера и параметры камеры (полученные при калибровке — см. раздел «Калибровка камеры» ниже), можно вычислить расстояние и ориентацию относительно маркера.

Определение координат

```python
import cv2
import numpy as np
from pioneer_sdk import Camera
import cv2.aruco as aruco
import threading
import time

def load_coefficients(path):
    cv_file = cv2.FileStorage(path, cv2.FILE_STORAGE_READ)
    camera_matrix = cv_file.getNode("mtx").mat()
    dist_coeffs = cv_file.getNode("dist").mat()
    cv_file.release()
    return camera_matrix, dist_coeffs

class PoseEstimatorThread:
    def __init__(self, camera_matrix, dist_coeffs):
        self.camera = Camera()
        self.running = True
        self.latest_frame = None
        self.processed_frame = None
        self.frame_lock = threading.Lock()

        self.camera_matrix = camera_matrix
        self.dist_coeffs = dist_coeffs

        # Dictionary of aruco-markers
        self.aruco_dict =
        ↪  aruco.getPredefinedDictionary(aruco.DICT_6X6_50)

        # Parameters for marker detection
        if hasattr(aruco, 'DetectorParameters_create'):
            self.parameters = aruco.DetectorParameters_create()
        else:
            self.parameters = aruco.DetectorParameters()
```

```
        self.size_of_marker = 0.05  # side length in meters

        # Coordinates of marker corners
        self.points_of_marker = np.array([
            (self.size_of_marker / 2, -self.size_of_marker / 2, 0),
            (-self.size_of_marker / 2, -self.size_of_marker / 2, 0),
            (-self.size_of_marker / 2, self.size_of_marker / 2, 0),
            (self.size_of_marker / 2, self.size_of_marker / 2, 0),
        ])

    def detect_markers_compat(self, image):
        if hasattr(aruco, 'ArucoDetector'):
            # Новый API (OpenCV 4.7+)
            detector = aruco.ArucoDetector(self.aruco_dict,
            ↪  self.parameters)
            return detector.detectMarkers(image)
        else:
            # Старый API
            return aruco.detectMarkers(image, self.aruco_dict,
            ↪  parameters=self.parameters)

    def capture_thread(self):
        while self.running:
            frame = self.camera.get_cv_frame()
            if frame is not None:
                with self.frame_lock:
                    self.latest_frame = frame.copy()
            time.sleep(0.01)

    def processing_thread(self):
        while self.running:
            frame_to_process = None
            with self.frame_lock:
                if self.latest_frame is not None:
                    frame_to_process = self.latest_frame.copy()

            if frame_to_process is None:
                time.sleep(0.01)
                continue

            # Detect markers
            corners, ids, rejected =
            ↪  self.detect_markers_compat(frame_to_process)

            if ids is not None and len(ids) > 0:
                # Calculate pose for first detected marker
                success, rvecs, tvecs = cv2.solvePnP(
                    self.points_of_marker, corners[0],
                    ↪  self.camera_matrix, self.dist_coeffs
                )
```

```
                if success:
                    # Draw axes
                    cv2.drawFrameAxes(frame_to_process,
                    ↪  self.camera_matrix, self.dist_coeffs, rvecs,
                    ↪  tvecs, 0.1)
                    # print(f"Marker Pose: {tvecs.T}") # Optional
                    ↪  logging

            with self.frame_lock:
                self.processed_frame = frame_to_process

    def run(self):
        print("Запуск оценки позы... Нажмите ESC для выхода.")

        t_capture = threading.Thread(target=self.capture_thread)
        t_process = threading.Thread(target=self.processing_thread)

        t_capture.daemon = True
        t_process.daemon = True

        t_capture.start()
        t_process.start()

        try:
            while True:
                display_frame = None
                with self.frame_lock:
                    if self.processed_frame is not None:
                        display_frame = self.processed_frame.copy()
                    elif self.latest_frame is not None:
                        display_frame = self.latest_frame.copy()

                if display_frame is not None:
                    cv2.imshow("Pose Estimation", display_frame)

                if cv2.waitKey(1) == 27:
                    self.running = False
                    break

                time.sleep(0.03)

        finally:
            self.running = False
            t_capture.join(timeout=1.0)
            t_process.join(timeout=1.0)
            cv2.destroyAllWindows()

if __name__ == "__main__":
    # Ensure data.yml exists or provide correct path
    try:
```

```
        camera_matrix, dist_coeffs = load_coefficients("data.yml")
        estimator = PoseEstimatorThread(camera_matrix, dist_coeffs)
        estimator.run()
    except Exception as e:
        print(f"Ошибка загрузки калибровки или запуска: {e}")
        print("Убедитесь, что файл 'data.yml' с параметрами калибровки
        ↪  существует.")
```

Посадка по маркеру

В этом примере дрон взлетает, занимает точку $(0, 0, 1)$ над стартом и висит, пока камера ищет маркеры. Как только в кадре появляется маркер с ID 42, дрон садится. Логика полёта работает в третьем потоке и только читает список `detected_ids`, который заполняет поток обработки.

Полет по маркеру

```
import cv2
import cv2.aruco as aruco
from pioneer_sdk import Pioneer, Camera
import threading
import time

class DroneFlightThread:
    def __init__(self):
        self.camera = Camera()
        self.drone = Pioneer()
        self.running = True
        self.latest_frame = None
        self.processed_frame = None
        self.frame_lock = threading.Lock()
        self.detected_ids = []

        # ArUco setup
        self.aruco_dict =
        ↪  aruco.getPredefinedDictionary(aruco.DICT_6X6_50)

        if hasattr(aruco, 'DetectorParameters_create'):
            self.parameters = aruco.DetectorParameters_create()
        else:
            self.parameters = aruco.DetectorParameters()

    def detect_markers_compat(self, image):
        if hasattr(aruco, 'ArucoDetector'):
            detector = aruco.ArucoDetector(self.aruco_dict,
            ↪  self.parameters)
            return detector.detectMarkers(image)
        else:
```

```
            return aruco.detectMarkers(image, self.aruco_dict,
            ↪  parameters=self.parameters)

    def capture_thread(self):
        while self.running:
            frame = self.camera.get_cv_frame()
            if frame is not None:
                with self.frame_lock:
                    self.latest_frame = frame.copy()
            time.sleep(0.01)

    def processing_thread(self):
        while self.running:
            frame_to_process = None
            with self.frame_lock:
                if self.latest_frame is not None:
                    frame_to_process = self.latest_frame.copy()

            if frame_to_process is None:
                time.sleep(0.01)
                continue

            gray = cv2.cvtColor(frame_to_process, cv2.COLOR_BGR2GRAY)
            corners, ids, rejected = self.detect_markers_compat(gray)

            current_ids = []
            if ids is not None:
                aruco.drawDetectedMarkers(frame_to_process, corners,
                ↪  ids)
                current_ids = ids.flatten().tolist()

            # Update shared state
            self.detected_ids = current_ids

            with self.frame_lock:
                self.processed_frame = frame_to_process

    def flight_logic(self):
        print("Арминг и взлет...")
        self.drone.arm()
        time.sleep(1) # Даем автопилоту перейти в режим готовности
        self.drone.takeoff()
        time.sleep(2)

        print("Летим в точку наблюдения...")
        self.drone.go_to_local_point(x=0, y=0, z=1, yaw=0)

        landing_marker_id = 42

        while self.running:
            if landing_marker_id in self.detected_ids:
```

```
                print(f"Посадочный маркер ({landing_marker_id}) найден!
                ↪  Сажусь...")
                self.drone.land()
                self.running = False # Stop other threads
                break

            time.sleep(0.1)

    def run(self):
        print("Запуск полетной миссии... Нажмите ESC для выхода.")

        t_capture = threading.Thread(target=self.capture_thread)
        t_process = threading.Thread(target=self.processing_thread)
        t_flight = threading.Thread(target=self.flight_logic)

        t_capture.daemon = True
        t_process.daemon = True
        t_flight.daemon = True # Flight logic runs in background

        t_capture.start()
        t_process.start()
        t_flight.start()

        try:
            while self.running:
                display_frame = None
                with self.frame_lock:
                    if self.processed_frame is not None:
                        display_frame = self.processed_frame.copy()
                    elif self.latest_frame is not None:
                        display_frame = self.latest_frame.copy()

                if display_frame is not None:
                    cv2.imshow('Pioneer Camera', display_frame)

                if cv2.waitKey(1) == 27:
                    print("Принудительная посадка...")
                    self.drone.land()
                    self.running = False
                    break

                time.sleep(0.03)

        finally:
            self.running = False
            # Wait a bit for threads to notice
            time.sleep(0.5)
            cv2.destroyAllWindows()
            self.drone.close_connection()

if __name__ == "__main__":
```

```
mission = DroneFlightThread()
mission.run()
```

8.2. Визуальная навигация

Зная координаты углов маркера на изображении, мы можем вычислить положение дрона относительно маркера. Это называется **Pose Estimation** (Оценка позы).

Определение положения (Pose Estimation)

Для точного определения расстояния и угла наклона используется функция `cv2.solvePnP` (в старых версиях OpenCV для этого была `cv2.aruco.estimatePoseSingleMarkers`). Она возвращает два вектора:

- **tvec (Translation Vector):** Вектор смещения (x, y, z) — где маркер относительно камеры.
- **rvec (Rotation Vector):** Вектор вращения — как маркер наклонен.

Для работы этой функции необходимо знать параметры камеры (матрицу калибровки), которые можно получить с помощью скрипта калибровки (раздел «Калибровка камеры» ниже).

Простая логика следования

Даже без сложной математики можно реализовать следование за маркером:

1. Найти центр маркера на изображении (среднее арифметическое углов).
2. Найти центр кадра (обычно 320x240 для разрешения 640x480).
3. Если $X_{marker} > X_{center}$, значит маркер справа $\rightarrow$ поворачиваем вправо.
4. Если $X_{marker} < X_{center}$, значит маркер слева $\rightarrow$ поворачиваем влево.

python/aruco_examples/12_follow_aruco.py

```
import cv2
import cv2.aruco as aruco
import numpy as np
from pioneer_sdk import Pioneer, Camera
import threading
import time
from collections import deque
```

```
def load_coefficients(path):
    try:
        cv_file = cv2.FileStorage(path, cv2.FILE_STORAGE_READ)
        camera_matrix = cv_file.getNode("mtx").mat()
        dist_coeffs = cv_file.getNode("dist").mat()
        cv_file.release()
        return camera_matrix, dist_coeffs
    except Exception:
        print("Warning: Calibration file not found, using dummy
        ↪  values.")
        return np.eye(3), np.zeros((5, 1))

class FollowArucoThread:
    def __init__(self, camera_matrix, dist_coeffs):
        self.camera = Camera()
        self.drone = Pioneer()
        self.running = True
        self.latest_frame = None
        self.processed_frame = None
        self.frame_lock = threading.Lock()

        self.camera_matrix = camera_matrix
        self.dist_coeffs = dist_coeffs

        # Flight control variables
        self.latest_coordinates = None # [x, y, z] relative to camera
        self.x_center = None

        # ArUco setup
        self.aruco_dict =
        ↪  aruco.getPredefinedDictionary(aruco.DICT_4X4_50)

        if hasattr(aruco, 'DetectorParameters_create'):
            self.parameters = aruco.DetectorParameters_create()
        else:
            self.parameters = aruco.DetectorParameters()

        self.size_of_marker = 0.1 # m
        self.points_of_marker = np.array([
            (self.size_of_marker / 2, -self.size_of_marker / 2, 0),
            (-self.size_of_marker / 2, -self.size_of_marker / 2, 0),
            (-self.size_of_marker / 2, self.size_of_marker / 2, 0),
            (self.size_of_marker / 2, self.size_of_marker / 2, 0),
        ])

        # FPS calculation
        self.frame_times = deque(maxlen=30)
        self.last_fps_log_time = time.time()

    def detect_markers_compat(self, image):
        if hasattr(aruco, 'ArucoDetector'):
```

```
            detector = aruco.ArucoDetector(self.aruco_dict,
            ↪  self.parameters)
            return detector.detectMarkers(image)
        else:
            return aruco.detectMarkers(image, self.aruco_dict,
            ↪  parameters=self.parameters)

    def capture_thread(self):
        while self.running:
            frame = self.camera.get_cv_frame()
            if frame is not None:
                with self.frame_lock:
                    self.latest_frame = frame.copy()
            time.sleep(0.01)

    def processing_thread(self):
        while self.running:
            frame_to_process = None
            with self.frame_lock:
                if self.latest_frame is not None:
                    frame_to_process = self.latest_frame.copy()

            if frame_to_process is None:
                time.sleep(0.01)
                continue

            try:
                corners, ids, _ =
                ↪  self.detect_markers_compat(frame_to_process)

                if ids is not None and len(corners) > 0:
                    # Calculate center for visualization
                    x_c = int(sum([p[0] for p in corners[0][0]]) / 4)
                    y_c = int(sum([p[1] for p in corners[0][0]]) / 4)

                    # Draw visual feedback
                    dot_size = 5
                    frame_to_process[y_c - dot_size:y_c + dot_size, x_c
                    ↪  - dot_size:x_c + dot_size] = [0, 0, 255]
                    aruco.drawDetectedMarkers(frame_to_process, corners)

                    # Solve PnP
                    success, rvecs, tvecs = cv2.solvePnP(
                        self.points_of_marker, corners[0],
                        ↪  self.camera_matrix, self.dist_coeffs
                    )

                    if success:
                        self.latest_coordinates = [tvecs.item(0),
                        ↪  tvecs.item(1), tvecs.item(2)]
                        self.x_center = x_c
```

```
                    else:
                        self.latest_coordinates = None
                        self.x_center = None
                else:
                    self.latest_coordinates = None
                    self.x_center = None

                # FPS Logging
                now = time.time()
                self.frame_times.append(now)
                if now - self.last_fps_log_time > 2.0 and
                ↪  len(self.frame_times) > 1:
                    fps = len(self.frame_times) / (self.frame_times[-1]
                    ↪  - self.frame_times[0])
                    # print(f"[Video Thread] FPS: {fps:.2f}")
                    self.last_fps_log_time = now

                with self.frame_lock:
                    self.processed_frame = frame_to_process

            except cv2.error as e:
                print(f"OpenCV error: {e}")

    def control_loop(self):
        """Main control loop for following the marker"""
        # Note: This logic assumes we want to control the drone based on
        ↪  vision.
        # Implementation depends on specific requirements (e.g., keep
        ↪  marker in center).

        # Simple P-controller constants
        k_p_yaw = 0.1
        k_p_fwd = 0.2
        target_dist = 1.0

        while self.running:
            if self.latest_coordinates is not None:
                x, y, z = self.latest_coordinates
                dist = z # Z in camera frame is distance forward

                # Simple logic:
                # 1. Rotate to keep x near 0 (center of image
                ↪  horizontally)
                # 2. Move forward/back to keep z near target_dist

                yaw_cmd = 0 # Calculate based on x
                vx_cmd = 0  # Calculate based on z

                # Placeholder for actual control logic
                # self.drone.set_manual_speed(vx=vx_cmd, vy=0, vz=0,
                ↪  yaw_rate=yaw_cmd)
```

```
                pass

            time.sleep(0.05)

    def run(self):
        print("Запуск следования за маркером... Нажмите ESC для
        ↪  выхода.")

        t_capture = threading.Thread(target=self.capture_thread)
        t_process = threading.Thread(target=self.processing_thread)
        t_control = threading.Thread(target=self.control_loop)

        t_capture.daemon = True
        t_process.daemon = True
        t_control.daemon = True

        t_capture.start()
        t_process.start()
        t_control.start()

        try:
            while self.running:
                display_frame = None
                with self.frame_lock:
                    if self.processed_frame is not None:
                        display_frame = self.processed_frame.copy()
                    elif self.latest_frame is not None:
                        display_frame = self.latest_frame.copy()

                if display_frame is not None:
                    cv2.imshow('Follow ArUco', display_frame)

                if cv2.waitKey(1) == 27:
                    self.running = False
                    break

                time.sleep(0.03)

        finally:
            self.running = False
            time.sleep(0.5)
            cv2.destroyAllWindows()
            self.drone.close_connection()

if __name__ == "__main__":
    camera_matrix, dist_coeffs = load_coefficients("data.yml")
    follower = FollowArucoThread(camera_matrix, dist_coeffs)
    follower.run()
```

Этот пример — заготовка следования. Поток обработки ищет маркер (словарь `DICT_4X4_50`), рисует точку в его центре и через `cv2.solvePnP` вычис-

ляет смещение маркера относительно камеры. Метод `control_loop` пока ничего не отправляет дрону: дрон не взлетает, а команды движения закомментированы. Дописать регулятор по пунктам списка выше — ваша задача (о пропорциональном регуляторе — в разделе «Трекинг человека и жестов»).

Задание 8.2: Посадка на базу

Реализуйте алгоритм: дрон висит в воздухе. Если он видит маркер ID 1, он летит вперед. Если видит маркер ID 2, летит назад. Если видит маркер ID 0 — садится.

8.3. Продвинутая работа с камерой

Мы уже использовали камеру для поиска маркеров. Теперь разберемся, как работать с видеопотоком более гибко.

Класс VideoStream

Для получения «сырого» видеопотока (без автоматического декодирования в OpenCV формат, что иногда нужно для оптимизации) используется класс `VideoStream`.

python/camera_examples/camera_stream.py

```python
from pioneer_sdk import VideoStream

if __name__ == "__main__":
    stream = VideoStream()      # Создаем объект потока видео

    while True:                 # Бесконечный цикл, ожидающий команд
        cmd = input()           # Ожидаем ввода команды от пользователя
        ↪  в консоль

        if cmd == "start":      # Если введена команда 'start'
            stream.start()      # Запускаем видеопоток

        elif cmd == "stop":     # Если введена команда 'stop'
            stream.stop()       # Останавливаем видеопоток
```

Этот код запускает передачу видео по Wi-Fi. Вы можете открыть этот поток в плеере VLC (адрес обычно `rtsp://192.168.4.1:8554/front`). Как и вся эта часть курса, RTSP-трансляция требует реального дрона: браузерный симулятор не отдает поток наружу, открыть его в VLC не получится.

Захват кадров и фотографирование

Чтобы сделать фото, нужно получить текущий кадр и сохранить его в файл с помощью `cv2.imwrite`.

Задание 8.3: Фотограф

Напишите программу, которая показывает видео с дрона на экране. При нажатии клавиши `S` (Save) текущий кадр должен сохраняться в файл `photo_N.jpg`, где N — порядковый номер фото.

Калибровка камеры

Для точных измерений (например, расстояния до маркера) нужно знать оптические параметры камеры: матрицу камеры и коэффициенты дисторсии. Процесс калибровки: 1. Распечатать шахматную доску (Chessboard). 2. Сделать серию снимков доски под разными углами. 3. Запустить скрипт калибровки (`camera_examples/camera_calibration.py`). Он вычислит параметры и сохранит их в файл `data.yml`: матрица камеры — узел `mtx`, коэффициенты дисторсии — узел `dist`. Этот же файл читают примеры с маркерами из предыдущих разделов.

Зачем это нужно?

Без калибровки прямые линии на изображении могут казаться изогнутыми (эффект «рыбьего глаза»), что приведет к ошибкам в навигации.

8.4. Телеметрия и работа с датчиками

У Pioneer есть множество датчиков. Мы научимся получать от них данные.

Основные датчики

- **Барометр:** Измеряет высоту.
- **IMU (Гироскоп + Акселерометр):** Измеряет углы наклона и ускорения.
- **Дальномер (ToF):** Лазерный датчик расстояния (смотрит вниз), измеряет дистанцию до пола.

- **Система оптического потока (OptiFlow):** Камера, смотрящая вниз, для удержания позиции.

Чтение данных дальномера

Дальномер полезен для точного удержания высоты при посадке или полете над препятствиями.

python/get_telemetry/GET_dist_sensor_data.py

```python
from pioneer_sdk import Pioneer
import time

# Создаем объект mini класса Pioneer
mini = Pioneer()

# Бесконечный цикл для постоянного опроса дальномера
while True:
    # Получаем данные с дальномера (в метрах)
    distance = mini.get_dist_sensor_data()

    # Проверяем, получены ли данные (не None)
    if distance is not None:
        # Выводим расстояние в консоль
        print(f'Расстояние: {distance:.2f} м')

    # Задержка вывода сообщений в консоль
    time.sleep(2)
```

Локальная позиция

Дрон постоянно вычисляет свои координаты (x, y, z) относительно точки старта. Эти данные можно получить через `get_local_position_lps()`.

Задание 8.4: Черный ящик

Напишите программу, которая во время полета каждые 0.5 секунды записывает в текстовый файл `flight_log.txt` текущее время, высоту и заряд батареи.

Задание 8.5: Мягкая посадка

Используя дальномер, реализуйте алгоритм мягкой посадки: дрон снижается со скоростью 0.5 м/с. Когда высота становится меньше 0.3 метра, скорость снижения уменьшается до 0.1 м/с. Снижение задаётся командой `set_manual_speed` с отрицательной `vz` (раздел 7.6).

8.5. Трекинг человека и жестов

Используя библиотеку **MediaPipe** от Google, мы можем распознавать позу человека (скелет) в реальном времени даже на обычном процессоре. Напомним: `mediapipe` в браузерном симуляторе недоступна, поэтому примеры этой главы запускаются только на компьютере с реальным дроном.

Как это работает?

1. Получаем кадр с камеры. 2. MediaPipe находит ключевые точки (суставы): плечи, локти, запястья, нос и т.д. 3. Анализируя взаимное расположение точек, мы можем определить жест.

Пример: Управление жестами

В примере `human-tracking.py` реализована сложная логика управления. Давайте разберем основные идеи.

Определение жеста «Руки вверх»: Если запястья (y_{wrist}) выше носа (y_{nose}), значит человек поднял руки.

Логика жеста

```
if (left_wrist.y < nose.y) and (right_wrist.y < nose.y):
    print("Жест: Взлет!")
    drone.arm()          # сначала моторы, как в главе про полёт
    time.sleep(1)
    drone.takeoff()
```

Пропорциональное управление (P-регулятор)

Чтобы дрон поворачивался за человеком, мы вычисляем ошибку: насколько центр человека смещен от центра кадра.

$$Error = X_{person} - X_{center}$$

Управляющее воздействие (скорость поворота) пропорционально ошибке:

$$YawRate = K_p \times Error$$

Где K_p — коэффициент чувствительности.

Задание 8.6: Джедай

Напишите упрощенную программу:

1. Дрон стоит на земле (не взлетает).
2. Камера включена.
3. Если вы поднимаете левую руку — дрон зажигает левый светодиод красным.
4. Если правую — правый зеленым.
5. Если скрещиваете руки (расстояние между запястьями мало) — мигает всеми цветами.

8.6. Групповой полет (Рой дронов)

Управление роем (Swarm) — сложная, но захватывающая задача. Главное здесь — синхронизация и единая система координат.

Архитектура роя

Мы будем использовать централизованную архитектуру:

- **Один компьютер (Наземная станция)** управляет всеми дронами.
- Каждый дрон имеет свой уникальный IP-адрес.
- Программа создает отдельный поток (`Thread`) для каждого дрона.

Пример: Синхронный взлет

В файле `group_flight_examples/swarm_OPT.py` показан пример, где несколько дронов выполняют задачи параллельно.

Ключевые моменты: 1. Создаем список IP-адресов. 2. Для каждого IP создаем объект `Pioneer`. Адрес обязательно передается по имени — `Pioneer(ip=ip)`: первый позиционный параметр конструктора называется `name`, поэтому запись `Pioneer(ip)` задаст дрону имя, а подключение пойдет на адрес по умолчанию. 3. Используем `threading.Thread` для запуска функций управления одновременно.

Псевдокод роя

```
drones = [Pioneer(ip=ip1), Pioneer(ip=ip2)]

def mission(drone):
    drone.arm()
    time.sleep(1)   # пауза между arm и takeoff обязательна
    drone.takeoff()
    time.sleep(5)
    drone.land()

# Запуск потоков
threads = []
for d in drones:
    t = threading.Thread(target=mission, args=(d,))
    t.start()
    threads.append(t)

# Ожидание завершения
for t in threads:
    t.join()
```

Безопасность роя

При групповых полетах риск столкновения возрастает многократно! 1. Всегда проверяйте координаты старта. 2. Расстояние между дронами должно быть не менее 1.5 метров. 3. Держите палец на кнопке аварийного отключения (или имейте общий скрипт `disarm_all`).

Задание 8.7: Танец

Напишите программу для двух дронов. Дрон А взлетает. Дрон Б ждет 5 секунд, затем взлетает. Дрон А садится. Дрон Б садится. Используйте `time.sleep` и потоки.

8.7. Итоговые проекты курса

Поздравляем! Вы освоили управление дроном Pioneer на языке Python. Теперь пришло время применить знания в комплексных проектах.

Проект 1: Автономный инспектор (Склад)

Задача: Дрон должен облететь 4 стеллажа (маркеры ID 0, 1, 2, 3), расположенных по углам комнаты. **Алгоритм:** 1. Взлет. 2. Поиск маркера 0. Подлет к нему на 1 метр. 3. Поворот на 90 градусов, поиск маркера 1. 4. ... 5. Возврат на точку старта и посадка.

Проект 2: «Следуй за мной» (Follow Me)

Задача: Дрон должен удерживать человека в центре кадра на безопасном расстоянии. **Требования:** 1. Использовать MediaPipe для детекции человека. 2. Если человек подходит близко (плечи занимают весь кадр) — дрон отлетает назад. 3. Если человек уходит — дрон летит вперед. 4. Поворот за смещением человека влево/вправо.

Проект 3: Гонка по точкам

Задача: Пролететь через полосу препятствий (ворота), отмеченные ArUco-маркерами, как можно быстрее. **Усложнение:** Маркеры могут быть расположены на разной высоте. Дрон должен пролетать сквозь ворота (или над ними).

Критерии оценки проектов

1. **Стабильность:** Программа не падает с ошибками.
2. **Безопасность:** Дрон не совершает резких опасных маневров.
3. **Читаемость кода:** Код разбит на функции, есть комментарии.
4. **Функциональность:** Задача выполнена полностью.

Часть III.

Профессиональные навигационные системы

9. Теоретические основы визуальной навигации

9.1. Введение в ArUco и AprilTag

Визуальная навигация — это способ определения положения робота в пространстве с помощью анализа изображений. В отличие от GPS, который работает только на улице и имеет погрешность в несколько метров, визуальные маркеры позволяют дрону позиционироваться с точностью до сантиметра даже в закрытых помещениях.

ArUco (Augmented Reality University of Cordoba) и **AprilTag** — это самые популярные библиотеки фидуциальных (опорных) маркеров. Они представляют собой черно-белые квадраты, которые содержат закодированную информацию (ID маркера) и позволяют алгоритмам компьютерного зрения однозначно определить ориентацию маркера в пространстве.

9.2. Принцип работы алгоритма обнаружения

Процесс обнаружения маркера на изображении состоит из нескольких этапов. Понимание этого конвейера (pipeline) важно для диагностики проблем, когда дрон «не видит» маркеры.

9.2.1. 1. Предварительная обработка (Preprocessing)

Камера передает цветное изображение (RGB). Алгоритм сначала переводит его в оттенки серого (Grayscale), так как цвет не несет информации для черно-белых маркеров.

Затем применяется **адаптивная бинаризация** (Adaptive Thresholding). В отличие от простого порога (где все пиксели ярче 127 становятся белыми, а

темнее — черными), адаптивный метод вычисляет порог для каждой небольшой области изображения. Это позволяет находить маркеры даже при неравномерном освещении (например, когда одна часть маркера в тени).

Пример бинаризации (OpenCV)

```
import cv2

# ... (получение кадра frame) ...
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

# Адаптивный порог: (src, maxValue, adaptiveMethod, thresholdType,
↪  blockSize, C)
binary = cv2.adaptiveThreshold(gray, 255,
    cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
    cv2.THRESH_BINARY_INV, 7, 2)
```

9.2.2. 2. Поиск контуров (Contour Extraction)

Алгоритм ищет границы связных областей белых пикселей на черном фоне. Из всех найденных контуров отбираются только те, которые:

- Являются замкнутыми.
- Аппроксимируются четырехугольником (имеют 4 вершины).
- Являются выпуклыми (Convex).
- Имеют достаточный размер (площадь).

9.2.3. 3. Исправление перспективы (Perspective Warping)

Найденный четырехугольник может быть искажен перспективой (выглядеть как трапеция или ромб), если камера смотрит на него под углом. Алгоритм вычисляет матрицу гомографии, чтобы «выпрямить» изображение и получить идеальный квадрат (Canonical Marker Image).

9.2.4. 4. Декодирование (Decoding)

Выпрямленный квадрат разбивается на сетку (например, 6×6 клеток). Каждая клетка анализируется: черная — 0, белая — 1. Полученная матрица битов сверяется со словарем (Dictionary).

9.3. Коррекция ошибок и Расстояние Хэмминга

Маркеры ArUco спроектированы так, чтобы быть устойчивыми к ошибкам считывания. Даже если несколько пикселей (битов) будут распознаны неверно (из-за блика или грязи), алгоритм все равно сможет узнать маркер.
Это достигается за счет избыточности информации. Каждый маркер имеет параметр **Hamming Distance** (Расстояние Хэмминга) — минимальное количество битов, на которое отличаются любые два маркера в словаре.

- Если словарь имеет Hamming Distance = 4, система может обнаружить до 3 ошибок и исправить до 1 ошибки.
- Чем больше размер словаря (например, 6×6 против 4×4), тем больше информации можно закодировать и тем выше устойчивость к ложным срабатываниям.

9.4. Сравнение типов словарей (Dictionaries)

При выборе словаря нужно искать баланс между количеством доступных ID и размером маркера (читаемостью).

Словарь	Размер сетки	Кол-во ID	Применение
DICT_4X4_50	4×4 бита	50	Дальние дистанции, низкое разрешение. Меньше деталей — легче распознать.
DICT_5X5_1000	5×5 бит	1000	Стандарт для большинства задач.
DICT_6X6_250	6×6 бит	250	Высокая надежность, используется в картах Пионера и в симуляторе; его первые 50 маркеров — это `DICT_6X6_50`.
DICT_APRILTAG_36h11	6×6 бит	587	Промышленный стандарт, очень высокая устойчивость к ложным срабатываниям.

Таблица 2: Характеристики словарей маркеров

9.5. Математика позиционирования (PnP Problem)

После того как маркер найден и идентифицирован, нужно определить положение камеры относительно него. Эта задача называется **Perspective-n-Point (PnP)**.

Мы знаем:

1. Координаты 4 углов маркера в его собственной системе координат (3D object points): $(0,0,0), (S,0,0), (S,S,0), (0,S,0)$, где S — размер маркера.
2. Координаты этих же 4 углов на изображении камеры (2D image points) в пикселях.
3. Внутренние параметры камеры (Camera Matrix и Distortion Coefficients), полученные при калибровке.

Функция `cv2.solvePnP` решает систему уравнений и возвращает два вектора:

- `rvec` (Rotation Vector) — вектор вращения (в формате Родригеса), показывающий наклон камеры.
- `tvec` (Translation Vector) — вектор смещения (x, y, z), показывающий позицию маркера относительно центра камеры.

Важно

Важно! Вектор `tvec` показывает координаты *маркера в системе координат камеры*. Чтобы получить координаты *дрона в системе координат маркера* (для навигации), нужно инвертировать эту трансформацию:

$$R = Rodrigues(rvec)$$

$$Position_{drone} = -R^T \cdot tvec$$

9.6. Практические задания

Задание 9.1: Задание 1: Анализ словарей

Сгенерируйте маркер ID=1 из словарей 4x4, 5x5 и 6x6 одного физического размера (например, 200 пикселей). 1. Распечатайте их. 2. Отойдите с камерой на максимальное расстояние. 3. Какой маркер перестанет распознаваться первым? Почему?

Задание 9.2: Задание 2: Влияние освещения

Запустите скрипт обнаружения маркера. 1. Постепенно затемняйте комнату. На каком уровне освещенности маркер пропадет? 2. Посветите фонариком прямо на маркер (создайте блик). Как это влияет на бинаризацию (посмотрите окно с ч/б изображением)?

Контрольные вопросы

1. Почему для навигации не используются QR-коды? (Подсказка: сравните сложность детектирования угла наклона).
2. Что такое «адаптивная бинаризация» и зачем она нужна?
3. Если дрон видит маркер под углом 45 градусов, как алгоритм понимает, что это квадрат, а не ромб?
4. В чем физический смысл вектора `tvec`?

10. Создание навигационных карт

10.1. Калибровка камеры

Перед тем как создавать карту, необходимо откалибровать камеру. Камера дрона — это не идеальный оптический прибор. Линзы (особенно широкоугольные «рыбий глаз») вносят искажения: прямые линии на краях кадра могут выглядеть изогнутыми. Это называется **дисторсией** (Distortion).
Если не учесть эти искажения, координаты маркеров будут вычислены с ошибкой в десятки сантиметров.

10.1.1. Процесс калибровки

Для калибровки используется специальный шаблон — шахматная доска (Chessboard).

1. Распечатайте шаблон (например, 9×6 клеток) на жестком листе.
2. Запустите скрипт калибровки (обычно доступен в примерах OpenCV).
3. Покажите доску камере под разными углами (20-30 кадров).
4. Скрипт вычислит:
   - **Camera Matrix**: Фокусное расстояние (f_x, f_y) и оптический центр (c_x, c_y).
   - **Distortion Coefficients**: Коэффициенты радиальной (k_1, k_2, k_3) и тангенциальной (p_1, p_2) дисторсии.

Скрипт ниже, как и `camera_calibration.py` из раздела про камеру, сохраняет параметры в `data.yml` (узлы `mtx` и `dist`); примеры навигации загружают их оттуда при запуске. Шаблон `camera_params.yaml` из приложения показывает тот же набор чисел в формате, который принимают сторонние инструменты.

Скрипт калибровки камеры

```
#!/usr/bin/python3

"""
Performs camera calibration using checkerboard pattern. Saves
 ↪  calibration
values into a file.
More: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html
"""
```

```
import cv2
import numpy as np
import glob
import time
from pioneer_sdk import Camera


def get_images_from_folder(folder_name, file_type="*.jpg"):
    """
    Retreive images from a directory using a provided path
    ↪  (`folder_name`)
    """
    images_list = glob.glob(folder_name + "/" + file_type)
    images = []
    for fname in images_list:
        img = cv2.imread(fname)
        images.append(img)
    return images


def get_images_from_drone_camera(camera, save=False):
    """
    Retreive images from a drone's camera
    """
    images = []
    print("Press '1' to take frame\nPress ESC or 'q' to finish")
    while True:
        frame = camera.get_cv_frame()
        if frame is not None:
            cv2.imshow("frames", frame)

            key = cv2.waitKey(1)
            if key == 27 or key == ord("q"):
                cv2.destroyAllWindows()
                return images
            if key == ord("1"):
                images.append(frame)
                print("Frame #" + str(len(images)))
                if save:
                    cv2.imwrite("frame_" + str(len(images)) + ".jpg",
                    ↪  frame)
                time.sleep(1)


def get_images_from_computer_camera(capture, save=False):
    """
    Retreive images using the PC's camera device
    """
    images = []
    print("Press '1' to take frame\nPress ESC or 'q' to finish")
```

```
    while True:
        ret, frame = capture.read()
        cv2.imshow("frames", frame)

        key = cv2.waitKey(1)
        if key == 27 or key == ord("q"):
            cv2.destroyAllWindows()
            return images
        if key == ord("1"):
            images.append(frame)
            print("Frame #" + str(len(images)))
            if save:
                cv2.imwrite("frame_" + str(len(images)) + ".jpg", frame)
            time.sleep(1)


def calibrate(images):
    # Defining the dimensions of checkerboard
    CHECKERBOARD = (6, 9)
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30,
    ↪  0.001)

    # Creating vector to store vectors of 3D points for each
    ↪  checkerboard image
    objpoints = []
    # Creating vector to store vectors of 2D points for each
    ↪  checkerboard image
    imgpoints = []

    # Defining the world coordinates for 3D points
    objp = np.zeros((1, CHECKERBOARD[0] * CHECKERBOARD[1], 3),
    ↪  np.float32)
    objp[0, :, :2] = np.mgrid[0 : CHECKERBOARD[0], 0 :
    ↪  CHECKERBOARD[1]].T.reshape(-1, 2)

    # Extracting path of individual image stored in a given directory
    for img in images:
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        # Find the chess board corners
        # If desired number of corners are found in the image then ret =
        ↪  true
        ret, corners = cv2.findChessboardCorners(
            gray,
            CHECKERBOARD,
            cv2.CALIB_CB_ADAPTIVE_THRESH
            + cv2.CALIB_CB_FAST_CHECK
            + cv2.CALIB_CB_NORMALIZE_IMAGE,
        )

        if ret == True:
            objpoints.append(objp)
```

```
            # refining pixel coordinates for given 2d points.
            corners2 = cv2.cornerSubPix(gray, corners, (11, 11), (-1,
            ↪  -1), criteria)

            imgpoints.append(corners2)

            # Draw and display the corners
            img = cv2.drawChessboardCorners(img, CHECKERBOARD, corners2,
            ↪  ret)

        cv2.imshow("img", img)
        cv2.waitKey(0)

    cv2.destroyAllWindows()

    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(
        objpoints, imgpoints, gray.shape[::-1], None, None
    )

    return mtx, dist


def save_coefficients(mtx, dist, path):
    """Save the camera matrix and the distortion coefficients to given
    ↪  path/file."""
    cv_file = cv2.FileStorage(path, cv2.FILE_STORAGE_WRITE)
    cv_file.write("mtx", mtx)
    cv_file.write("dist", dist)
    # note you *release* you don't close() a FileStorage object
    cv_file.release()


def load_coefficients(path):
    """Loads camera matrix and distortion coefficients."""
    # FILE_STORAGE_READ
    cv_file = cv2.FileStorage(path, cv2.FILE_STORAGE_READ)

    # note we also have to specify the type to retrieve other wise we
    ↪  only get a
    # FileNode object back instead of a matrix
    camera_matrix = cv_file.getNode("mtx").mat()
    dist_coeffs = cv_file.getNode("dist").mat()

    cv_file.release()
    return camera_matrix, dist_coeffs


def main():
    """
    В этом примере изображения берутся с камеры дрона.
    Однако вы можете изменить источник изображений, заменив
```

```
    функцию `get_images_from_drone_camera` на другую, например:

    ```
    # Загрузить изображения из папки
    images = get_images_from_folder('images')
    # ...
    ```

    ```
    # Получить изображения с веб-камеры компьютера
    capture = cv2.VideoCapture(0)
    images = get_images_from_computer_camera(capture)
    capture.release()
    # ...
    ```
    """

    # Acquire images from the drone's camera
    camera = Camera()
    images = get_images_from_drone_camera(camera)

    mtx, dist = calibrate(images)

    save_coefficients(mtx, dist, "data.yml")

if __name__ == "__main__":
    main()
```

10.2. Генерация и печать маркеров

Используйте скрипт `19_map_creation_example2_generation.py` для создания изображений.

Генерация маркеров ArUco/AprilTag

```
import cv2
import cv2.aruco as aruco
import numpy as np
import os

# Create directory for output if it doesn't exist
output_dir = "../../../images/markers"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Define dictionaries to generate
dictionaries = {
```

```
    "DICT_4X4_50": aruco.DICT_4X4_50,
    "DICT_5X5_100": aruco.DICT_5X5_100,
    "DICT_6X6_250": aruco.DICT_6X6_250,
    "DICT_7X7_1000": aruco.DICT_7X7_1000,
    "DICT_APRILTAG_36h11": aruco.DICT_APRILTAG_36h11
}

def add_visual_aids(image, size_pixels):
    """
    Adds coordinate axes and grid to the marker image for educational
    ↪ purposes.
    """
    # Convert to color to add colored axes
    color_img = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)

    # Draw border
    border_size = int(size_pixels * 0.1)
    color_img = cv2.copyMakeBorder(color_img, border_size, border_size,
    ↪ border_size, border_size,
                                   cv2.BORDER_CONSTANT, value=[255, 255,
                                   ↪ 255])

    new_size = color_img.shape[0]
    center = new_size // 2

    # Draw Axes (X - Red, Y - Green)
    # Origin is top-left of the marker content (inside border)
    origin_x = border_size
    origin_y = border_size

    # X Axis
    cv2.arrowedLine(color_img, (origin_x, origin_y), (origin_x + 100,
    ↪ origin_y), (0, 0, 255), 2)
    cv2.putText(color_img, "X", (origin_x + 110, origin_y + 10),
    ↪ cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)

    # Y Axis
    cv2.arrowedLine(color_img, (origin_x, origin_y), (origin_x, origin_y
    ↪ + 100), (0, 255, 0), 2)
    cv2.putText(color_img, "Y", (origin_x + 10, origin_y + 110),
    ↪ cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

    return color_img

def generate_marker(dict_name, marker_id, size_pixels=200,
↪ visual_aids=False):
    """
    Generates a single ArUco marker image.
    """
    try:
```

```
        aruco_dict =
        ↪  aruco.getPredefinedDictionary(dictionaries[dict_name])

        # Draw the marker
        marker_image = np.zeros((size_pixels, size_pixels),
        ↪  dtype=np.uint8)
        marker_image = aruco.generateImageMarker(aruco_dict, marker_id,
        ↪  size_pixels)

        if visual_aids:
            final_image = add_visual_aids(marker_image, size_pixels)
            suffix = "_viz"
        else:
            final_image = marker_image
            suffix = ""

        # Save the marker
        filename = f"{output_dir}/{dict_name}_id{marker_id}{suffix}.png"
        cv2.imwrite(filename, final_image)
        print(f"Generated: {filename}")
    except Exception as e:
        print(f"Error generating marker {dict_name} ID {marker_id}:
        ↪  {e}")

def generate_json_metadata(dict_name, count, size_pixels):
    import json

    # Infer bits from name (e.g. DICT_4X4_50 -> 4)
    bits = 0
    if "4X4" in dict_name: bits = 4
    elif "5X5" in dict_name: bits = 5
    elif "6X6" in dict_name: bits = 6
    elif "7X7" in dict_name: bits = 7
    elif "APRILTAG" in dict_name: bits = "variable"

    data = {
        "dictionary": dict_name,
        "bits": bits,
        "count": count,
        "size_pixels": size_pixels,
        "description": "Marker set for Geoskan Pioneer"
    }
    filename = f"{output_dir}/{dict_name}_metadata.json"
    with open(filename, 'w') as f:
        json.dump(data, f, indent=4)
    print(f"Metadata saved: {filename}")

def generate_batch(dict_name, start_id, count, size_pixels=200,
↪  visual_aids=False):
    """
    Generates a batch of markers.
```

```
    """
    print(f"--- Generating {count} markers for {dict_name} ---")
    for i in range(start_id, start_id + count):
        generate_marker(dict_name, i, size_pixels, visual_aids)

    generate_json_metadata(dict_name, count, size_pixels)

if __name__ == "__main__":
    # 1. Standard sets
    generate_batch("DICT_4X4_50", 0, 5)
    generate_batch("DICT_5X5_100", 0, 2)
    generate_batch("DICT_6X6_250", 0, 5) # Standard for Pioneer
    generate_batch("DICT_7X7_1000", 0, 2)
    generate_batch("DICT_APRILTAG_36h11", 0, 5)

    # 2. Visualization examples (with axes)
    generate_marker("DICT_6X6_250", 0, 300, visual_aids=True)
    generate_marker("DICT_APRILTAG_36h11", 0, 300, visual_aids=True)

    # 3. Full range examples (uncomment to generate large sets)
    # generate_batch("DICT_6X6_250", 0, 50)
    # generate_batch("DICT_APRILTAG_36h11", 0, 50)

    print(f"Done! Markers saved in '{output_dir}' folder.")
```

10.2.1. Векторные маркеры

Для печати в типографии или создания масштабируемых карт рекомендуется использовать векторные форматы (PDF/SVG). Следующий скрипт генерирует маркеры с использованием векторной графики.

Генерация векторных маркеров

```
import cv2
import cv2.aruco as aruco
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import numpy as np
import os

# Define output directory
output_dir = "../../../images/markers/vector"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Helper function to generate raw bits of a marker
def get_marker_bits(dictionary, marker_id, side_pixels):
    """
```

```
    Generates a marker image with the exact pixel size matching its bit
    ↪ grid.
    For ArUco 4x4, the grid is 4x4 + 1 bit border = 6x6.
    For ArUco 6x6, the grid is 6x6 + 1 bit border = 8x8.
    For AprilTag 36h11, the grid is 8x8 + 1 bit border = 10x10.
    """
    # Create a small image (e.g. 10x10) to get raw bits
    # generateImageMarker scales up, but if we request small size it
    ↪ should work
    # OpenCV's generateImageMarker takes sidePixels as argument.
    # If sidePixels is small, it returns the raw bits.
    img = aruco.generateImageMarker(dictionary, marker_id, side_pixels)
    return img

def draw_vector_marker(ax, img, x_offset=0, y_offset=0, size=1.0):
    """
    Draws a marker on a matplotlib axis using Rectangle patches (Vector
    ↪ format).
    """
    rows, cols = img.shape
    pixel_size = size / rows

    # Draw white background
    rect = patches.Rectangle((x_offset, y_offset), size, size,
    ↪ linewidth=0, facecolor='white')
    ax.add_patch(rect)

    # Draw black pixels
    for r in range(rows):
        for c in range(cols):
            if img[r, c] == 0:
                # In matplotlib, origin is bottom-left usually, but we
                ↪ can invert axis
                # Let's assume standard image coordinates (0,0 is
                ↪ top-left) and invert Y later or now
                # Drawing from top-left (y = size - ...)
                rect_x = x_offset + c * pixel_size
                rect_y = y_offset + (rows - 1 - r) * pixel_size

                rect = patches.Rectangle((rect_x, rect_y), pixel_size,
                ↪ pixel_size, linewidth=0, facecolor='black')
                ax.add_patch(rect)

def save_single_marker(dict_name, marker_id, bit_size, filename):
    dictionary = aruco.getPredefinedDictionary(getattr(aruco,
    ↪ dict_name))
    img = get_marker_bits(dictionary, marker_id, bit_size)

    fig, ax = plt.subplots(figsize=(4, 4))
    draw_vector_marker(ax, img, size=10)
```

```
    ax.set_xlim(0, 10)
    ax.set_ylim(0, 10)
    ax.axis('off')
    ax.set_aspect('equal')

    plt.savefig(filename, format='pdf', bbox_inches='tight',
    ↪  pad_inches=0)
    plt.close()
    print(f"Saved vector marker: {filename}")

def save_grid_map(dict_name, rows, cols, bit_size, filename):
    dictionary = aruco.getPredefinedDictionary(getattr(aruco,
    ↪  dict_name))

    # A4 Page size approx (8.27 x 11.69 inches)
    fig, ax = plt.subplots(figsize=(8.27, 11.69))

    margin_x = 1
    margin_y = 1
    cell_width = (8.27 - 2 * margin_x) / cols
    cell_height = (11.69 - 2 * margin_y) / rows
    marker_size = min(cell_width, cell_height) * 0.8

    # Start drawing from top-left
    start_x = margin_x
    start_y = 11.69 - margin_y - cell_height

    for i in range(rows):
        for j in range(cols):
            marker_id = i * cols + j
            img = get_marker_bits(dictionary, marker_id, bit_size)

            x = start_x + j * cell_width + (cell_width - marker_size) /
            ↪  2
            y = start_y - i * cell_height + (cell_height - marker_size)
            ↪  / 2

            draw_vector_marker(ax, img, x_offset=x, y_offset=y,
            ↪  size=marker_size)

            # Add label
            ax.text(x + marker_size/2, y - 0.2, f"ID: {marker_id}",
            ↪  ha='center', va='top', fontsize=8)

    ax.set_xlim(0, 8.27)
    ax.set_ylim(0, 11.69)
    ax.axis('off')
    ax.set_aspect('equal')

    plt.savefig(filename, format='pdf', bbox_inches='tight')
    plt.close()
```

```
    print(f"Saved vector grid: {filename}")

if __name__ == "__main__":
    # 1. Single Markers
    # ArUco 4x4 (Size 6x6 pixels)
    save_single_marker("DICT_4X4_50", 0, 6,
    ↪  f"{output_dir}/aruco_4x4_id0.pdf")
    # AprilTag 36h11 (Size 10x10 pixels usually? Let's verify)
    # AprilTag 36h11 is 8x8 data + 1 bit border = 10x10 total
    save_single_marker("DICT_APRILTAG_36h11", 0, 10,
    ↪  f"{output_dir}/apriltag_id0.pdf")

    # 2. Grid Maps (4x4 = 16 markers)
    save_grid_map("DICT_4X4_50", 4, 4, 6,
    ↪  f"{output_dir}/aruco_grid_4x4.pdf")
    save_grid_map("DICT_APRILTAG_36h11", 4, 4, 10,
    ↪  f"{output_dir}/apriltag_grid_4x4.pdf")
```

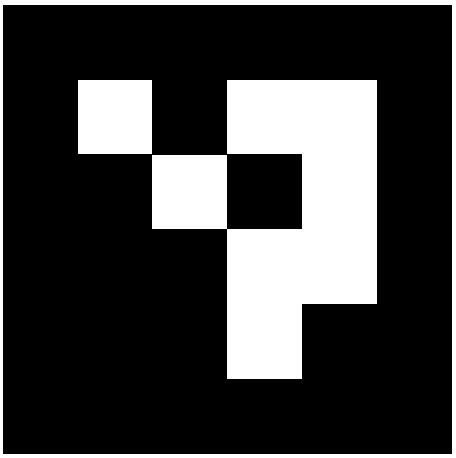


Рис. 5: ArUco 4x4 (Вектор)

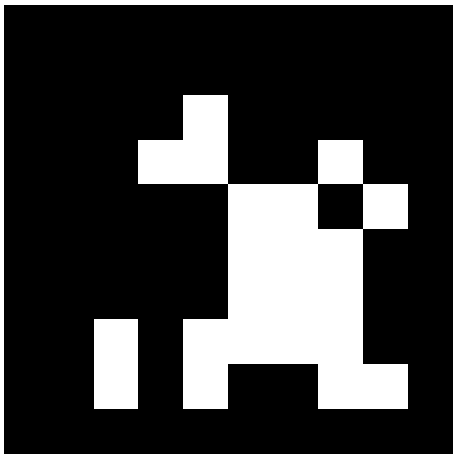


Рис. 6: AprilTag 36h11 (Вектор)

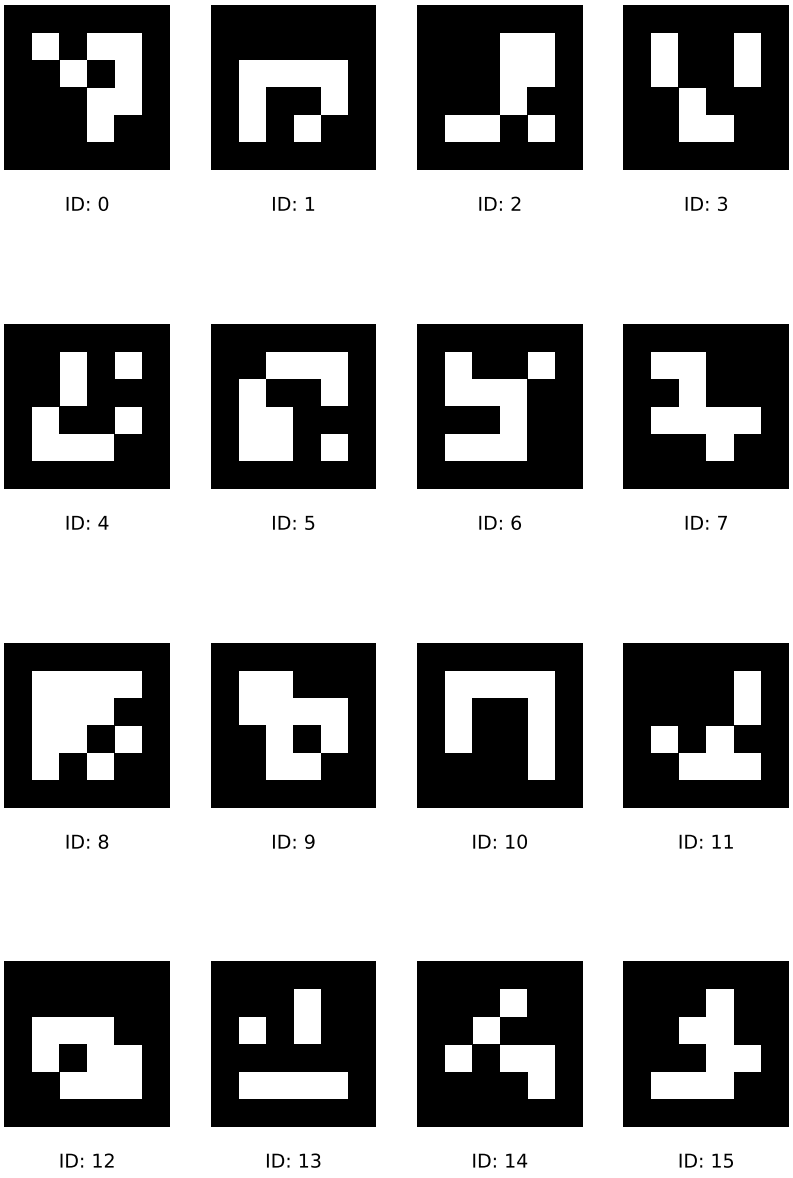


Рис. 7: Карта маркеров 4x4 (ArUco, вектор)

Внимание

Важно! При печати в настройках принтера выберите «Реальный размер» (Actual Size) или «Масштаб 100%». Иначе размер маркера изменится, и навигация будет врать.

10.2.2. Требования к монтажу

1. **Жесткость:** Маркер должен быть идеально плоским. Изгиб бумаги даже на 5 мм приведет к ошибке определения угла (Yaw/Pitch/Roll) на 10-15 градусов. Наклейте маркер на картон, пластик или фанеру.
2. **Quiet Zone:** Вокруг черной рамки маркера обязательно должно быть белое поле шириной не менее одного бита. Если наклеить черный маркер на черный пол, он не распознается.
3. **Матовое покрытие:** Глянцевая бумага или скотч создают блики от ламп. В месте блика камера видит белое пятно вместо черного пикселя. Используйте матовую ламинацию или матовый скотч.
4. **Освещение:** Избегайте размещения маркеров в глубокой тени или напротив ярких окон (контражур).

10.3. Форматы карт (JSON, XML, KML)

Карта маркеров — это файл, описывающий положение каждого маркера. Наш инструмент `19_map_creation_example3_builder.py` поддерживает несколько форматов.

Инструмент создания карт

```
import json
import math
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.path import Path
import numpy as np

class Marker:
    def __init__(self, marker_id, size, x, y, z, yaw=0,
    ↪  dictionary="DICT_6X6_250"):
        self.id = marker_id
        self.size = size  # in meters
        self.x = x        # local x (meters)
        self.y = y        # local y (meters)
        self.z = z        # local z (meters) - height
        self.yaw = yaw    # orientation (degrees)
        self.dictionary = dictionary

    def to_dict(self):
        return {
            "id": self.id,
```

```
            "size": self.size,
            "position": {"x": self.x, "y": self.y, "z": self.z},
            "orientation": {"yaw": self.yaw},
            "dictionary": self.dictionary
        }

class MarkerMap:
    def __init__(self, name="MyMap", origin_gps=(0, 0)):
        self.name = name
        self.origin_lat = origin_gps[0]
        self.origin_lon = origin_gps[1]
        self.markers = []

    def add_marker(self, marker):
        self.markers.append(marker)

    def generate_grid(self, rows, cols, step_x, step_y, start_id=0,
    ↪  marker_size=0.2, dictionary="DICT_4X4_50"):
        """
        Generates a grid of markers.
        """
        for i in range(rows):
            for j in range(cols):
                # Calculate position
                # Assuming grid starts at (0,0) and extends in +X and
                ↪  +Y
                x = i * step_x
                y = j * step_y
                z = 0  # On the floor

                marker_id = start_id + i * cols + j

                self.add_marker(Marker(
                    marker_id=marker_id,
                    size=marker_size,
                    x=x,
                    y=y,
                    z=z,
                    yaw=0,
                    dictionary=dictionary
                ))
        print(f"Generated grid: {rows}x{cols}, {len(self.markers)}
        ↪  markers.")

    def save_yaml(self, filename):
        """
        Exports map to YAML format (Clever 4 style).
        """
        import yaml

        # Prepare data structure for ROS/Clever
```

```
        # Usually it's a list of markers with id, size, frame_id, pose
        markers_data = []
        for m in self.markers:
            markers_data.append({
                "id": int(m.id),
                "size": float(m.size),
                "frame_id": "map", # Global frame
                "pose": {
                    "position": {
                        "x": float(m.x),
                        "y": float(m.y),
                        "z": float(m.z)
                    },
                    "orientation": {
                        # Simplified quaternion for yaw rotation around
                        ↪  Z
                        "x": 0.0,
                        "y": 0.0,
                        "z": math.sin(math.radians(m.yaw)/2),
                        "w": math.cos(math.radians(m.yaw)/2)
                    }
                }
            })

        with open(filename, 'w') as f:
            yaml.dump({"markers": markers_data}, f,
            ↪  default_flow_style=False)
        print(f"Map YAML saved to {filename}")

    def save_png_grid(self, filename, resolution_px=4096):
        """
        Exports the map as a high-resolution PNG grid (Clever 4 style).
        """
        import cv2
        import cv2.aruco as aruco

        # Determine map bounds
        xs = [m.x for m in self.markers]
        ys = [m.y for m in self.markers]
        min_x, max_x = min(xs), max(xs)
        min_y, max_y = min(ys), max(ys)

        # Add padding (e.g., 1 meter)
        padding_m = 1.0
        width_m = (max_x - min_x) + 2 * padding_m
        height_m = (max_y - min_y) + 2 * padding_m

        # Calculate scale (pixels per meter)
        # We want the largest dimension to fit in resolution_px
        scale = resolution_px / max(width_m, height_m)
```

```
        # Create white image
        img_h = int(height_m * scale)
        img_w = int(width_m * scale)
        image = np.ones((img_h, img_w), dtype=np.uint8) * 255

        print(f"Generating PNG map: {img_w}x{img_h} pixels (scale:
        ↪  {scale:.1f} px/m)")

        # Draw markers
        for m in self.markers:
            # Get marker dictionary
            try:
                # Map string dictionary name to cv2.aruco constant
                if hasattr(aruco, m.dictionary):
                    adict = aruco.getPredefinedDictionary(getattr(aruco,
                    ↪  m.dictionary))
                else:
                    # Fallback to default
                    adict =
                    ↪  aruco.getPredefinedDictionary(aruco.DICT_6X6_250)

                # Generate marker image
                marker_size_px = int(m.size * scale)
                marker_img = aruco.generateImageMarker(adict, m.id,
                ↪  marker_size_px)

                # Calculate position in image
                # Transform (x,y) from center-based to top-left image
                ↪  coordinates
                # Image origin is top-left. Map origin (0,0) is
                ↪  somewhere in the middle.
                # x_img = (m.x - min_x + padding) * scale
                # y_img = (max_y - m.y + padding) * scale (invert Y for
                ↪  image coords if needed, but typically maps align
                ↪  X-right Y-up vs image X-right Y-down)

                # Let's assume standard map: X right, Y up. Image: X
                ↪  right, Y down.
                # So we flip Y.

                center_x_px = int((m.x - min_x + padding_m) * scale)
                center_y_px = int((max_y - m.y + padding_m) * scale) #
                ↪  Invert Y

                top_left_x = center_x_px - marker_size_px // 2
                top_left_y = center_y_px - marker_size_px // 2

                # Place marker
                if 0 <= top_left_y < img_h - marker_size_px and 0 <=
                ↪  top_left_x < img_w - marker_size_px:
```

```
                    image[top_left_y:top_left_y+marker_size_px,
                    ↪  top_left_x:top_left_x+marker_size_px] =
                    ↪  marker_img

                    # Add simple text ID for verification (optional,
                    ↪  might clutter)
                    # cv2.putText(image, str(m.id), (top_left_x,
                    ↪  top_left_y - 10), cv2.FONT_HERSHEY_SIMPLEX, 1,
                    ↪  0, 2)

            except Exception as e:
                print(f"Error drawing marker {m.id}: {e}")

        cv2.imwrite(filename, image)
        print(f"Map PNG saved to {filename}")

    def save_json(self, filename):
        data = {
            "map_name": self.name,
            "origin": {"lat": self.origin_lat, "lon": self.origin_lon},
            "markers": [m.to_dict() for m in self.markers]
        }
        with open(filename, 'w') as f:
            json.dump(data, f, indent=4)
        print(f"Map saved to {filename}")

    def save_pdf(self, filename, trajectory=None):
        """
        Exports the map to a PDF file for printing/visualization.
        Includes marker positions, IDs, and optional trajectory.
        """
        fig, ax = plt.subplots(figsize=(11.69, 8.27))  # A4 Landscape
        ↪  (inches)

        # Plot markers
        x_coords = []
        y_coords = []

        for m in self.markers:
            x_coords.append(m.x)
            y_coords.append(m.y)

            # Draw marker as a square
            # m.x, m.y is the center of the marker
            half_size = m.size / 2
            rect = patches.Rectangle(
                (m.x - half_size, m.y - half_size),
                m.size, m.size,
                linewidth=1, edgecolor='black', facecolor='none'
            )
            ax.add_patch(rect)
```

```
            # Add ID text
            ax.text(m.x, m.y, str(m.id), ha='center', va='center',
            ↪  fontsize=8, fontweight='bold')

            # Add orientation arrow (small)
            # Arrow pointing in yaw direction
            arrow_len = m.size * 0.4
            dx = arrow_len * math.cos(math.radians(m.yaw))
            dy = arrow_len * math.sin(math.radians(m.yaw))
            ax.arrow(m.x, m.y, dx, dy, head_width=m.size*0.1,
            ↪  head_length=m.size*0.1, fc='blue', ec='blue', alpha=0.5)

        # Plot trajectory if provided
        if trajectory:
            traj_x = [p[0] for p in trajectory]
            traj_y = [p[1] for p in trajectory]
            ax.plot(traj_x, traj_y, 'r--', label='Drone Trajectory',
            ↪  linewidth=2, alpha=0.7)

            # Mark start and end
            ax.scatter(traj_x[0], traj_y[0], c='green', marker='o',
            ↪  label='Start')
            ax.scatter(traj_x[-1], traj_y[-1], c='red', marker='x',
            ↪  label='End')
            ax.legend()

        # Set plot limits with some padding
        if x_coords and y_coords:
            ax.set_xlim(min(x_coords) - 1, max(x_coords) + 1)
            ax.set_ylim(min(y_coords) - 1, max(y_coords) + 1)

        ax.set_aspect('equal')
        ax.set_xlabel('X (meters)')
        ax.set_ylabel('Y (meters)')
        ax.set_title(f'Marker Map: {self.name}')
        ax.grid(True, linestyle=':', alpha=0.6)

        # Add metadata text
        info_text = f"Total Markers: {len(self.markers)}\nOrigin:
        ↪  {self.origin_lat}, {self.origin_lon}"
        plt.figtext(0.02, 0.02, info_text, fontsize=10)

        plt.savefig(filename, format='pdf')
        plt.close()
        print(f"Map visualization saved to {filename}")

    def save_kml(self, filename):
        """
        Simple KML export for Google Earth.
```

```
        Note: This uses a rough approximation for meters to degrees (1
        ↪  deg lat ~ 111km).
        For production, use a proper geodetic library like 'pyproj'.
        """
        kml_header = """<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Document>
    <name>{name}</name>
"""
        kml_footer = """  </Document>
</kml>"""

        kml_body = ""

        # Approximate conversion (very rough, valid only for small
        ↪  areas)
        lat_deg_per_meter = 1 / 111132.928
        lon_deg_per_meter = 1 / (111132.928 *
        ↪  math.cos(math.radians(self.origin_lat)))

        for m in self.markers:
            # Calculate GPS position from local offset
            m_lat = self.origin_lat + (m.x * lat_deg_per_meter)
            m_lon = self.origin_lon + (m.y * lon_deg_per_meter)

            kml_body += f"""
    <Placemark>
      <name>Marker {m.id}</name>
      <description>Size: {m.size}m, Yaw: {m.yaw}</description>
      <Point>
        <coordinates>{m_lon},{m_lat},{m.z}</coordinates>
      </Point>
    </Placemark>"""

        with open(filename, 'w') as f:
            f.write(kml_header.format(name=self.name) + kml_body +
            ↪  kml_footer)
        print(f"KML saved to {filename}")

    def save_xml(self, filename):
        """
        Generic XML export.
        """
        xml_content = f"""<?xml version="1.0" encoding="UTF-8"?>
<map>
    <name>{self.name}</name>
    <origin>
        <lat>{self.origin_lat}</lat>
        <lon>{self.origin_lon}</lon>
    </origin>
    <markers>
```

```
"""
        for m in self.markers:
            xml_content += f"""        <marker id="{m.id}">
            <size>{m.size}</size>
            <position>
                <x>{m.x}</x>
                <y>{m.y}</y>
                <z>{m.z}</z>
            </position>
            <orientation>
                <yaw>{m.yaw}</yaw>
            </orientation>
            <dictionary>{m.dictionary}</dictionary>
        </marker>
"""
        xml_content += """    </markers>
</map>"""

        with open(filename, 'w') as f:
            f.write(xml_content)
        print(f"XML saved to {filename}")

if __name__ == "__main__":
    # Example: Create a grid map for a flight zone (Clever 4 Style)
    # Origin: Let's assume a field coordinate
    my_map = MarkerMap("Clever_Map_16x16", origin_gps=(59.9343,
    ↪  30.3351)) # St. Petersburg center approx

    # Generate a 16x16 grid with 0.5m spacing, marker size 0.2m
    # Dictionary for Clever 4 is usually 4x4_50 or 6x6_250 depending on
    ↪  setup. Let's stick to 4x4_50 as per common educational kits or
    ↪  6x6_250 for Pioneer.
    # User request mentions "0-999 for ArUco", which implies larger
    ↪  dictionary. But usually grid maps use a subset.
    # Let's use DICT_4X4_50 as it is common for small grids, or extend
    ↪  if needed. 16x16 = 256 markers. DICT_4X4_50 has only 50 markers!
    # We need a larger dictionary. DICT_6X6_250 has 250 markers (almost
    ↪  enough, need 256).
    # DICT_5X5_1000 or DICT_6X6_1000 is safer. Pioneer standard is
    ↪  DICT_6X6_50 (only 50!).
    # If we need 16x16=256 unique markers, we MUST use a larger
    ↪  dictionary.
    # Let's use DICT_6X6_1000 to be safe and cover 0-255 IDs.

    my_map.generate_grid(rows=16, cols=16, step_x=0.5, step_y=0.5,
    ↪  start_id=0, marker_size=0.2, dictionary="DICT_6X6_1000")

    # Define a dummy trajectory (square path)
    trajectory = [
        (0.5, 0.5, 1.0),
        (3.5, 0.5, 1.0),
```

```
        (3.5, 3.5, 1.0),
        (0.5, 3.5, 1.0),
        (0.5, 0.5, 1.0)
    ]

    # Export standard formats
    my_map.save_json("../../data/maps/field_map.json")
    my_map.save_pdf("../../data/maps/field_map.pdf",
    ↪  trajectory=trajectory)

    # Export Clever 4 style artifacts
    my_map.save_yaml("../../data/maps/clever_map.yaml")
    my_map.save_png_grid("../../data/maps/clever_map_4096.png",
    ↪  resolution_px=4096)

    # Save config
    config = {
        "grid": {
            "rows": 16,
            "cols": 16,
            "step_x": 0.5,
            "step_y": 0.5
        },
        "marker": {
            "size": 0.2,
            "dictionary": "DICT_6X6_1000"
        }
    }
    with open("../../data/maps/map_config.json", 'w') as f:
        json.dump(config, f, indent=4)
        print("Config saved to ../../data/maps/map_config.json")

    # my_map.save_kml("field_map.kml") # Optional
    # my_map.save_xml("field_map.xml") # Optional
```

10.3.1. JSON (JavaScript Object Notation)

Основной формат для роботов. Легко читается программами на Python/C++.

Пример карты JSON

```
{
  "map_name": "Warehouse_Zone_A",
  "origin": {"lat": 59.9, "lon": 30.3},
  "markers": [
    {
      "id": 10,
      "size": 0.2,
      "position": {"x": 5.0, "y": 0.0, "z": 1.5},
```

```
      "orientation": {"yaw": 90.0}
    }
  ]
}
```

10.3.2. XML (eXtensible Markup Language)

Более строгий формат, часто используется в промышленных системах.

Пример карты XML

```xml
<map>
    <name>Warehouse_Zone_A</name>
    <marker id="10">
        <size>0.2</size>
        <position>
            <x>5.0</x> <y>0.0</y> <z>1.5</z>
        </position>
    </marker>
</map>
```

10.3.3. KML (Keyhole Markup Language)

Формат для отображения на географических картах (Google Earth, QGIS). Позволяет визуализировать схему расположения маркеров поверх спутниковых снимков. Это полезно для планирования полетов на улице.

10.4. Визуализация и планирование

Для больших полигонов рекомендуется сначала создать план в CAD-системе или нарисовать схему на миллиметровке, определить координаты установки маркеров, и только потом заносить их в файл карты.

Задание 10.1: Задание 3: Калибровка

Проведите процедуру калибровки камеры Pioneer. 1. Сделайте 20 снимков шахматной доски. 2. Запустите скрипт калибровки. 3. Сравните изображение до и после устранения дисторсии (функция `cv2.undistort`). Прямые ли линии косяков дверей на краях кадра?

Задание 10.2: Задание 4: Создание полигона

Создайте карту для полета «восьмеркой». 1. Нарисуйте траекторию на полу мелом. 2. Разместите 2 маркера в центрах петель восьмерки. 3. Измерьте их координаты рулеткой относительно точки старта. 4. Создайте JSON файл карты.

Контрольные вопросы

1. Что такое дисторсия и почему она мешает навигации?
2. Почему нельзя ламинировать маркеры глянцевой пленкой?
3. Зачем нужно белое поле вокруг маркера?
4. В чем преимущество формата KML перед JSON?

11. Алгоритмы комплексной навигации

11.1. Проблема шума и фильтрация

Данные, получаемые от системы компьютерного зрения, всегда содержат шум. Координаты дрона «дрожат» даже если он висит неподвижно. Это происходит из-за дискретности матрицы камеры, изменений освещения и вибраций корпуса.

Если передавать эти «сырые» координаты напрямую в полетный контроллер, дрон будет дергаться, пытаясь отработать каждое ложное смещение.

11.1.1. Фильтр Калмана (Kalman Filter)

Это стандарт де-факто в робототехнике. Фильтр Калмана не просто усредняет значения, он предсказывает, где дрон *должен* оказаться в следующий момент времени, и корректирует это предсказание на основе новых измерений.

Алгоритм работает циклически в два этапа:

1. **Prediction (Предсказание):** На основе физической модели (скорость, ускорение) мы оцениваем новую позицию. При этом растет неопределенность (мы не уверены в прогнозе).
2. **Update (Коррекция):** Мы получаем измерение от камеры. Сравниваем его с прогнозом. Чем точнее датчик (меньше шум измерения R), тем больше мы верим ему. Чем точнее наша модель (меньше шум процесса Q), тем больше верим прогнозу.

Пример 1D фильтра Калмана

```python
# См. полный код в examples/advanced_navigation/kalman_filter.py
def update(self, measurement):
    # 1. Prediction
    x_pred = self.x
    P_pred = self.P + self.Q

    # 2. Update
    K = P_pred / (P_pred + self.R) # Kalman Gain
    self.x = x_pred + K * (measurement - x_pred)
    self.P = (1 - K) * P_pred
    return self.x
```

Выше приведен только ключевой метод `update()`. Полный запускаемый скрипт с классом `KalmanFilter1D` и модельным экспериментом находится

в файле `advanced_navigation/kalman_filter.py` — именно его нужно запускать в заданиях ниже.

11.2. Системы координат

Одна из самых частых ошибок новичков — путаница в системах координат.

- **Body Frame (Связанная система):** Ось X всегда смотрит вперед (нос дрона), Y — влево/вправо, Z — вверх. Если дрон поворачивается, оси поворачиваются вместе с ним.
- **World Frame / NED (North-East-Down):** Ось X смотрит на Север, Y — на Восток, Z — Вниз (к земле). Эта система неподвижна. Локальная система Пионера, в которой мы задавали точки (раздел 5), тоже неподвижна, но устроена иначе: X вперёд от старта, Y влево, Z *вверх*. Если дрон на старте смотрит на север, при переходе между ними меняются знаки Y и Z.
- **Marker Frame:** Ось Z обычно смотрит перпендикулярно плоскости маркера (вверх от него).

Для навигации нам нужно переводить координаты маркера (полученные камерой в Body Frame) в глобальные координаты дрона (World Frame).

$$Pos_{world} = Pos_{drone} + R_{drone} \cdot Pos_{marker_relative}$$

11.3. Точная посадка с PID-регулятором

Для автоматической посадки на зарядную станцию используется PID-регулятор.

- **P (Proportional):** Чем дальше мы от центра, тем сильнее наклоняемся к нему.
- **I (Integral):** Если нас сносит ветром, мы накапливаем ошибку и наклоняемся сильнее, чтобы компенсировать снос.
- **D (Derivative):** Если мы летим к центру слишком быстро, мы «тормозим» заранее, чтобы не пролететь мимо.

Задание 11.1: Задание 5: Сравнение фильтров

Запустите скрипт `kalman_filter.py`. 1. Измените `measurement_noise` (шум измерения) с 0.1 до 5.0. Как ведет себя фильтр? 2. Сравните работу фильтра

Калмана и простого скользящего среднего (Moving Average) при резком скачке координат (например, дрон толкнули). Кто быстрее реагирует?

Задание 11.2: Задание 6: Посадка в круг

Напишите скрипт, который: 1. Ищет маркер с высоты 1.5 метра. 2. Снижается до 0.5 метра, удерживая маркер в центре кадра (используя P-регулятор по скоростям V_x, V_y — команда `set_manual_speed`, раздел 7.6). 3. Если ошибка позиционирования < 5 см в течение 2 секунд — совершает посадку.

Контрольные вопросы

1. Зачем нужен фильтр Калмана, если можно просто взять среднее арифметическое за 10 кадров?
2. Что произойдет с PID-регулятором, если перепутать знак у коэффициента P?
3. В какой системе координат (Body или World) камера выдает вектор `tvec`?

12. Техническая реализация программных модулей

В этой главе мы разберем исходный код инструментов, которые используются для создания карт и навигации.

12.1. Генератор карт (Marker Map Builder)

Скрипт `19_map_creation_example3_builder.py` (глава «Создание навигационных карт») позволяет создавать цифровые карты полигонов и экспортировать их в форматы JSON, XML и KML.

12.1.1. Структура класса Marker

Каждый маркер описывается классом `Marker`, который хранит:

- `id`: Уникальный номер маркера.
- `size`: Физический размер стороны в метрах.
- `x, y, z`: Локальные координаты относительно начала координат карты.
- `yaw`: Угол поворота (ориентация).
- `dictionary`: Тип словаря (например, DICT_6X6_250).

Класс Marker

```python
class Marker:
    def __init__(self, marker_id, size, x, y, z, yaw=0,
    ↪  dictionary="DICT_6X6_250"):
        self.id = marker_id
        self.size = size  # in meters
        self.x = x        # local x (meters)
        self.y = y        # local y (meters)
        self.z = z        # local z (meters) - height
        self.yaw = yaw    # orientation (degrees)
        self.dictionary = dictionary
```

12.1.2. Экспорт в XML

Метод `save_xml` генерирует структуру, понятную большинству промышленных систем. Мы используем f-строки для формирования XML-тегов.

Экспорт в XML

```
def save_xml(self, filename):
    xml_content = f"""<?xml version="1.0" encoding="UTF-8"?>
<map>
    <name>{self.name}</name>
    <origin>
        <lat>{self.origin_lat}</lat>
        <lon>{self.origin_lon}</lon>
    </origin>
    <markers>
"""
    for m in self.markers:
        xml_content += f"""         <marker id="{m.id}">
            <size>{m.size}</size>
            <position>
                <x>{m.x}</x>
                <y>{m.y}</y>
                <z>{m.z}</z>
            </position>
            <orientation>
                <yaw>{m.yaw}</yaw>
            </orientation>
        </marker>
"""
    xml_content += """    </markers>
</map>"""
    with open(filename, 'w') as f:
        f.write(xml_content)
```

12.1.3. Экспорт в KML для Google Earth

Для отображения маркеров на глобусе Google Earth используется формат KML. Главная сложность здесь — перевод локальных метров (x, y) в глобальные градусы широты и долготы.

Мы используем приближенную формулу (для малых расстояний):

- 1 градус широты ≈ 111132 метра.
- 1 градус долготы $\approx 111132 \cdot \cos(\text{широта})$ метров.

Внимание

Точность GPS Данная формула имеет погрешность. Для профессиональной геодезии следует использовать библиотеку `pyproj` и проекцию UTM.

12.2. Модуль визуализации

Для проверки правильности карты полезно отрисовать её на 2D-плоскости перед полетом. Библиотека `matplotlib` позволяет построить схему расположения маркеров.

Визуализация карты

```
import matplotlib.pyplot as plt

def plot_map(self):
    x_coords = [m.x for m in self.markers]
    y_coords = [m.y for m in self.markers]
    ids = [m.id for m in self.markers]

    plt.figure(figsize=(10, 10))
    plt.scatter(x_coords, y_coords, c='red', marker='s', s=100)

    for i, txt in enumerate(ids):
        plt.annotate(f"ID {txt}", (x_coords[i], y_coords[i]), xytext=(5,
        ↪  5), textcoords='offset points')

    plt.grid(True)
    plt.axis('equal')
    plt.title(f"Map: {self.name}")
    plt.xlabel("X (meters)")
    plt.ylabel("Y (meters)")
    plt.show()
```

Задание 12.1: Задание 7: Свой парсер

Напишите функцию `load_json(filename)`, которая считывает файл карты и создает объекты класса `Marker`. Проверьте, что карта корректно загружается после сохранения.

13. Банк практических заданий

В этом разделе собраны задачи для закрепления материала. Задания разделены по темам и уровням сложности:

- **Базовый (Б):** Проверка понимания основных команд.
- **Продвинутый (П):** Требует написания алгоритмов.
- **Комплексный (К):** Мини-проект, объединяющий несколько тем.

13.1. Блок 1: Работа с камерой и ArUco (aruco_examples, camera_examples)

1. **(Б) «Фотограф»:** Напишите скрипт, который делает фото каждые 5 секунд и сохраняет их в папку с текущей датой и временем в названии.
2. **(Б) «Детектор движения»:** Сравнивайте два последовательных кадра. Если количество изменившихся пикселей превышает порог — выводите сообщение «Движение обнаружено!».
3. **(П) «Умный светофор»:** Распечатайте красный и зеленый круги. Если дрон видит зеленый цвет — он взлетает на 1 метр. Если красный — садится.
4. **(П) «Поиск маркера»:** Дрон взлетает и вращается вокруг своей оси (yaw). Как только он видит любой ArUco маркер — останавливается и «смотрит» на него 3 секунды.
5. **(П) «Дистанция»:** Дрон висит напротив маркера. Если маркер приближается (человек подходит с маркером в руках) — дрон отлетает назад, сохраняя дистанцию 1 метр.
6. **(К) «Инвентаризация»:** В комнате разложены 5 маркеров с ID от 0 до 4. Дрон должен облететь комнату, найти все маркеры и записать в файл: «ID 3 найден в 12:00», «ID 1 найден в 12:05».
7. **(К) «Посадка на платформу»:** Дрон должен найти маркер на полу, снизиться, зависнуть точно над центром и совершить посадку.

13.2. Блок 2: Телеметрия (get_telemetry)

1. **(Б) «Черный ящик»:** Записывайте высоту, заряд батареи и углы наклона (roll, pitch) в CSV файл 10 раз в секунду во время полета.

2. **(Б) «Сигнализация»:** Если напряжение батареи падает ниже 3.5 В на банку (как в задаче «Умная батарея»), дрон должен включить красную подсветку и начать посадку.
3. **(П) «Альтиметр»:** Взлетите и висите. Выводите разницу между высотой по барометру и высотой по дальномеру (ToF). В каких ситуациях они отличаются?
4. **(П) «Компас»:** Напишите программу, которая разворачивает дрон носом строго на Север (0 градусов), затем на Восток (90), Юг (180) и Запад (270).
5. **(К) «Виртуальная стена»:** Задайте границы зоны полета (например, X от -2 до 2 метров). Если телеметрия показывает выход за границы — дрон должен остановиться и вернуться в центр.

13.3. Блок 3: Миссии и полет по точкам (mission_examples)

1. **(Б) «Квадрат»:** Пролетите по точкам $(0,0) \to (1,0) \to (1,1) \to (0,1) \to (0,0)$ на высоте 1 метр.
2. **(Б) «Лестница»:** Пролетите вперед на 1 метр, набрав 0.5 метра высоты. Повторите 3 раза (как по ступенькам).
3. **(П) «Спираль»:** Напишите генератор точек для полета по спирали (расширяющийся круг с набором высоты).
4. **(П) «Патруль»:** Дрон летает между двумя точками бесконечно, пока не увидит препятствие (по датчику расстояния или камере).
5. **(К) «Сканирование поля»:** Реализуйте полет «змейкой» над зоной 3×3 метра для создания фотоплана местности.

13.4. Блок 4: Групповой полет (group_flight_examples)

1. **(Б) «Синхронный старт»:** Два дрона взлетают одновременно по команде с ПК, висят 5 секунд и садятся.
2. **(П) «Карусель»:** Два дрона висят напротив друг друга и меняются местами, пролетая по дуге (чтобы не столкнуться).
3. **(П) «Ведущий-ведомый»:** Первый дрон летит по программе (квадрат). Второй дрон получает телеметрию первого и повторяет его движения со смещением (на 1 метр сзади).
4. **(К) «Шоу дронов»:** Составьте хореографию для 4 дронов под музыку (ис-

пользуйте мигание светодиодами в такт).

13.5. Блок 5: Трекинг объектов (PioneerHumanTracking)

1. **(Б) «Приветствие»:** Если дрон видит лицо человека — он мигает зеленым и делает «кивок» (pitch вперед-назад).
2. **(П) «Следи за цветом»:** Возьмите яркий оранжевый мяч. Дрон должен поворачиваться (yaw) так, чтобы мяч всегда был в центре кадра.
3. **(П) «Держи дистанцию»:** Дрон видит лицо. Если лицо занимает слишком много места в кадре (человек близко) — отлетаем. Слишком мало (человек далеко) — подлетаем.
4. **(К) «Оператор»:** Дрон летит боком параллельно идущему человеку, снимая его профиль (требует сложной координации движений).

13.6. Критерии оценки выполнения

Задание считается выполненным, если:

1. Код запускается без синтаксических ошибок.
2. Дрон выполняет поставленную задачу в реальных условиях не менее 3 раз подряд.
3. Код содержит комментарии, объясняющие логику работы.
4. Соблюдены меры безопасности (дрон не вылетает за пределы зоны, экстренная посадка работает).

14. Типичные ошибки и их устранение

В процессе работы с дроном часто возникают ситуации, когда «код правильный, но ничего не работает». В этой главе собраны самые частые проблемы.

14.1. Проблемы с камерой и маркерами

14.1.1. Дрон не видит маркер

- **Причина 1: Плохое освещение.** Камере темно, выдержка увеличивается, появляется «смаз» (Motion Blur) при движении.
- **Решение:** Включите дополнительный свет. Маркер должен быть освещен равномерно.
- **Причина 2: Блики.** Маркер заламинирован глянцевой пленкой или лежит под лампой.
- **Решение:** Переместите маркер или используйте матовый скотч.
- **Причина 3: Неверный размер.** В коде указан размер 0.2 метра, а напечатан маркер 18 см.
- **Решение:** Измерьте черную рамку линейкой и исправьте константу в коде.
- **Причина 4: Quiet Zone.** Маркер вырезан по черному контуру без белого поля.
- **Решение:** Наклейте его на белый лист бумаги.

14.1.2. Дрон летит не туда

- **Причина 1: Перепутаны оси.** Вы думаете, что X — это вправо, а это вперед.
- **Решение:** Проверьте ориентацию (Body Frame). Ось X дрона смотрит вперед — туда же, куда нос и передняя камера; Y — влево, Z — вверх. Не путайте с системой самой камеры в OpenCV: там вперед «из объектива» смотрит ось Z (вектор `tvec`).
- **Причина 2: Ошибка калибровки.** Камера «рыбий глаз» сильно искажает картинку по краям.
- **Решение:** Проведите калибровку камеры заново.
- **Причина 3: Yaw (Рысканье).** Дрон повернулся на 90 градусов, но код продолжает отправлять команды в старой системе координат.

- **Решение:** Используйте глобальную систему координат (World Frame) или учитывайте текущий угол Yaw.

14.2. Проблемы с подключением и Wi-Fi

14.2.1. Connection Refused / Timeout

- **Причина 1:** Компьютер подключился к другой Wi-Fi сети (с интернетом).
- **Решение:** Проверьте, что вы подключены к точке доступа Pioneer_XXXX.
- **Причина 2:** Брандмауэр Windows блокирует порт 8000/8888.
- **Решение:** Разрешите Python доступ в частные сети.

14.3. Ошибки в коде Python

14.3.1. IndentationError

Python очень чувствителен к отступам. Не смешивайте табы и пробелы. Используйте 4 пробела для отступа.

14.3.2. OpenCV Error: Assertion failed

Обычно означает, что функция получила пустое изображение (None).

- Проверьте, что `camera.get_cv_frame()` не вернул None.
- Проверьте, что файл изображения существует (если читаете с диска).

14.4. Аварийные ситуации

14.4.1. Дрон не реагирует на Land

Если программная посадка не срабатывает:

1. Не паникуйте.
2. Нажмите кнопку Disarm в Ground Station (если есть).
3. Аккуратно поймайте дрон снизу за аккумулятор (берегите пальцы от пропеллеров!) и переверните его вверх ногами. Моторы отключатся автоматически.

Внимание

Внимание! Никогда не пытайтесь схватить дрон сверху или сбоку — пропеллеры вращаются со скоростью 10 000 об/мин и могут нанести серьезные травмы.

Часть IV.

Приложения

A. Справочник Lua API Пионера

Краткий справочник функций, которыми пользуется книга. Полное описание с примерами — в справочнике API симулятора (значок книги в заголовке редактора) и в документации Геоскан.

A.1. Команды автопилота: ap

`ap.push(Ev.MCE_PREFLIGHT)`	предстартовая подготовка и запуск моторов; ответ — `Ev.ENGINES_STARTED`
`ap.push(Ev.MCE_TAKEOFF)`	взлёт на высоту `Flight_com_takeoffAlt`; ответ — `Ev.TAKEOFF_COMPLETE`
`ap.push(Ev.MCE_LANDING)`	посадка; ответ — `Ev.COPTER_LANDED`
`ap.push(Ev.ENGINES_ARM)`	запуск моторов без взлёта (проверка)
`ap.push(Ev.ENGINES_DISARM)`	немедленное выключение моторов
`ap.goToLocalPoint(x, y, z [, t])`	полёт в точку (метры), необязательно — за `t` секунд; ответ — `Ev.POINT_REACHED`
`ap.updateYaw(rad)`	курс в радианах; `math.rad(deg)` для перевода

A.2. События: Ev в callback(event)

`Ev.ENGINES_STARTED`	моторы запущены после предстарта
`Ev.TAKEOFF_COMPLETE`	взлёт завершён
`Ev.POINT_REACHED`	точка `goToLocalPoint` достигнута
`Ev.POINT_DECELERATION`	дрон начал торможение перед точкой
`Ev.COPTER_LANDED`	дрон сел
`Ev.LOW_VOLTAGE1 / LOW_VOLTAGE2`	батарея разряжена: предупреждение / пора садиться
`Ev.SHOCK`	удар, столкновение
`Ev.CONTROL_FAIL`	потеря управления
`Ev.ENGINE_FAIL`	отказ мотора

A.3. Таймеры: Timer

- `Timer.callLater(delay, fn)` — однократный вызов через `delay` секунд.

- `t = Timer.new(period, fn)` — периодический таймер; `t:start()`, `t:stop()`.
- `time()` — секунды с запуска программы (для измерений).
- `sleep(t)` — блокирующая пауза; в полётных программах не используется.

A.4. Светодиоды: Ledbar

- `leds = Ledbar.new(29)` — линейка: 4 светодиода платы (`0..3`) и 25 модуля матрицы (`4..28`).
- `leds:set(i, r, g, b)` — цвет светодиода `i`, компоненты `0..1`.
- `Ledbar.fromHSV(h, s, v)` — перевод оттенка (градусы), насыщенности и яркости (проценты) в `r, g, b`.

A.5. Датчики: Sensors

`local x, y, z = Sensors.lpsPosition()`	положение в локальной системе, м
`local vx, vy, vz = Sensors.lpsVelocity()`	скорость, м/с
`local yaw = Sensors.lpsYaw()`	курс по системе позиционирования, рад
`local roll, pitch, yaw = Sensors.orientation()`	углы ориентации
`local ax, ay, az = Sensors.accel()`	ускорения, м/с^2
`local gx, gy, gz = Sensors.gyro()`	угловые скорости, рад/с
`local d = Sensors.range()`	дальномер: расстояние до поверхности под дроном, м
`local h = Sensors.altitude()`	высота по барометру, м
`local v = Sensors.battery()`	напряжение батареи, В
`local c1, ..., c8 = Sensors.rc()`	восемь каналов пульта, от -1 до 1; `table.pack(Sensors.rc())` — в таблицу

A.6. Идиомы курса

- Распаковка: `local unpack = table.unpack`; `leds:set(i, unpack(color))`; `ap.goToLocalPoint(unpack(points[i]))`.
- «Тернарный»: `local v = cond and A or B`.
- Счётчик шагов: `nextPoint()` отправляет `points[curr]` и увеличивает `curr`; вызывается из `TAKEOFF_COMPLETE` и `POINT_REACHED`.

- Конечный автомат: переменная `state` в начале файла (глобальная или `local`) и проверка `event == Ev.X and state == "Y"`; таймеры этапа проверяют `state` первой строкой.

B. Шпаргалка: Lua и Python

Одно и то же действие на двух языках. Lua ждёт событий в `callback`, Python опрашивает автопилот в цикле.

Действие	Lua	Python
Моторы	`ap.push(Ev.MCE_PREFLIGHT)`	`pioneer.arm()`
Взлёт	`ap.push(Ev.MCE_TAKEOFF)`	`pioneer.takeoff()`
Точка	`ap.goToLocalPoint(x, y, z)`	`pioneer.go_to_local_point(x=, y=, z=, yaw=)`
Ждать точку	`if event == Ev.POINT_REACHED`	`while not pioneer.point_reached(): time.sleep(0.1)`
Курс	`ap.updateYaw(рад)`	`yaw=` в `go_to_local_point`
Посадка	`ap.push(Ev.MCE_LANDING)`	`pioneer.land()`
Выключить моторы	`ap.push(Ev.ENGINES_DISARM)`	`pioneer.disarm()`
Пауза	`Timer.callLater(t, f)`	`time.sleep(t)`
Периодически	`Timer.new(p, f):start()`	цикл `while` с `time.sleep(p)`
Светодиоды	`leds:set(i, r, g, b)`, 0…1	`pioneer.led_control(led_id, r, g, b)`, 0…255
Позиция	`Sensors.lpsPosition()`	`pioneer.get_local_position_lps()`
Дальномер	`Sensors.range()`	`pioneer.get_dist_sensor_data()`
Батарея	`Sensors.battery()`	`pioneer.get_battery_status()`
Кадр камеры	`camera.requestMakeShot()`	`Camera().get_cv_frame()`
Вывод	`print(...)`	`print(...)`
Завершение	—	`pioneer.close_connection()`

Три главные мысли

- Дрон — автопилот: вы отдаёте команду и ждёте события, что она выполнена.
- В Lua ждут через `callback(event)`, в Python — опросом `while not ...`.
- Координаты — метры от старта, Z вверх, курс в радианах; их показывает `Ctrl`+щелчок в симуляторе.

С. Аппаратура управления

В этой главе мы подробно рассмотрим устройство и настройку аппаратуры радиоуправления FlySky FS-i6S, которая входит в стандартный комплект образовательного квадрокоптера «Геоскан Пионер». Пульт дистанционного управления (аппаратура) — это основной инструмент внешнего пилота, позволяющий контролировать полет, переключать режимы работы и выполнять настройку параметров дрона на расстоянии.

Аппаратура работает в паре с приемником, установленным на борту квадрокоптера, передавая команды по радиоканалу. Для корректной работы системы необходимо правильно настроить пульт и установить связь с приемником.

С.1. Настройка пульта

Перед первым использованием или после сброса настроек необходимо сконфигурировать пульт управления. Это обеспечит правильную интерпретацию команд полетным контроллером.

Для настройки пульта выполните следующие действия шаг за шагом:

1. Вставьте батарейки (4 шт. типа AA) в батарейный отсек и включите пульт дистанционного управления. Для включения необходимо нажать одновременно две кнопки питания (расположены в центре лицевой панели) и удерживать их до включения экрана.
2. После загрузки на дисплее появится главный экран с отображением текущего состояния, напряжения батарей и положения стиков.

Подсказка

Подсказка: В целях безопасности пульт проверяет положение переключателей при включении. Если на экране отображается предупреждение «Warning!!! Place all switches in their up position», это означает, что один из тумблеров находится в активном состоянии или стик газа не в нуле. Приведите все переключатели в верхнее положение (от себя), а левый стик (газ) опустите в самое нижнее положение.

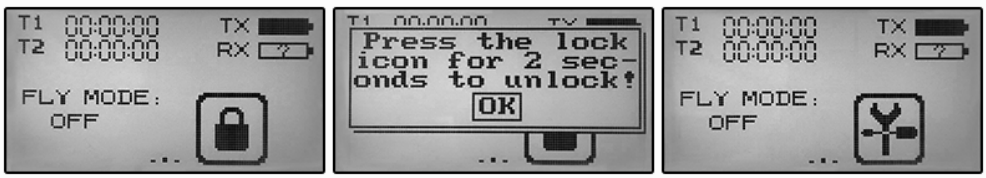


Рис. 8: Главный экран пульта управления FS-i6S

3. Изначально сенсорный экран заблокирован от случайных нажатий, о чём сигнализирует иконка замка в нижнем правом углу. Для разблокировки нажмите на иконку и удерживайте её в течении 2-4 секунд. После разблокировки иконка сменится на изображение гаечного ключа.
4. Нажмите на иконку с гаечным ключом, чтобы попасть в меню настроек. Меню разделено на две основные вкладки:
   - **FUNCTION** — здесь производится настройка функций самого пульта (каналы, реверсы и т.д.).
   - **SYSTEM** — отвечает за системные настройки связи и приемника.

C.1.1. Настройки вкладки FUNCTION

В этой вкладке мы настроим назначение каналов и направление их работы. Это критически важно для того, чтобы движения стиков и переключения тумблеров вызывали ожидаемую реакцию дрона.

Для работы с «Пионером» выполните следующие настройки:

1. **Настройка реверса каналов (Reverse):**
   - Перейдите в меню **REVERSE**.
   - Установите каналы **Ch2** (Тангаж) и **Ch4** (Рыскание) в положение **Rev** (Реверс). Остальные каналы оставьте в Norm.
2. **Настройка вспомогательных каналов (Aux. Channels):** Для управления режимами полета и дополнительными функциями необходимо назначить физические тумблеры на логические каналы пульта. Перейдите в меню **AUX. CHANNELS** и настройте каналы следующим образом:
   - **Channel 5:** Нажмите на кнопку выбора канала. В окне **CH TYPE** выберите **SWx** (переключатель). Затем нажмите на надпись SwA (по умолчанию) и в открывшемся списке выберите тумблер **SwC** (трехпозиционный переключатель справа). Этот канал обычно используется для смены режимов полета.

- **Channel 6:** Аналогично настройте на тумблер **SwD** (крайний правый переключатель).
- **Channel 7:** Настройте на тумблер **SwB** (трехпозиционный переключатель слева).
- **Channel 8:** Настройте на тумблер **SwA** (крайний левый переключатель).



Рис. 9: Меню настройки вспомогательных каналов

С.1.2. Настройка вкладки SYSTEM

В этой вкладке задаются параметры протокола передачи данных и режим работы стиков.

1. **Выбор протокола выхода (Output Mode):**
   - Перейдите в меню **OUTPUT MODE**.
   - Убедитесь, что в поле **Output** выбран протокол **PPM**. Полетный контроллер «Пионера» настроен на прием сигналов именно в этом формате. Также должен быть выбран режим **S.BUS** в поле Serial, если используется соответствующее подключение, но для базовой настройки PPM является основным.
2. **Режим стиков (Sticks Mode):**
   - Перейдите в меню **STICKS MODE**.
   - Выберите режим **M2 (Mode 2)**. В этом режиме левый стик отвечает за Газ (вверх/вниз) и Рыскание (влево/вправо), а правый — за Тангаж (вперед/назад) и Крен (влево/вправо). Это наиболее распространенная схема управления для квадрокоптеров.

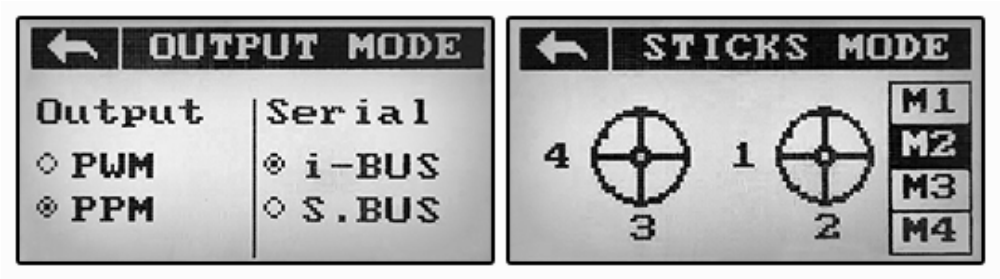


Рис. 10: Системные настройки: выбор режима PPM и Mode 2

С.2. Связь пульта с приемником

Процедура привязки (Binding) необходима для того, чтобы приемник, установленный на квадрокоптере, «узнал» ваш пульт и реагировал только на его команды. Это исключает пересечение сигналов при полетах нескольких дронов одновременно.

Внимание

Внимание: Для использования квадрокоптера «Пионер Мини» с пультом радиоуправления, в автопилот должны быть загружены соответствующие параметры! Без этого дрон может не реагировать на команды.

С.2.1. Процедура привязки

Выполните следующие действия для сопряжения устройств:

1. Убедитесь, что приемник (FS-A8S или аналогичный) подключен к плате автопилота.
2. Найдите на корпусе приемника маленькую кнопку с подписью **«BIND»**. Нажмите её и **удерживайте**.
3. **Не отпуская** кнопку BIND, подайте питание на квадрокоптер (подключите аккумулятор).
4. После подачи питания светодиод на приемнике должен начать **быстро мигать**. Это означает, что приемник перешел в режим привязки. Теперь кнопку можно отпустить.
5. Включите пульт управления. Зайдите в меню настроек (**SYSTEM**) и выберите пункт **RX Bind**.
6. На экране пульта появится надпись **Binding to RX...**. В этот момент пульт

ищет приемник в режиме привязки.

7. Как только связь установится, светодиод на приемнике начнет **мигать медленно**.
8. Для завершения процедуры нажмите иконку «Назад» (←) на пульте, чтобы выйти из меню RX Bind.
9. После выхода из режима привязки светодиод на приемнике должен загореться **постоянным светом** (красным). Это подтверждает успешное сопряжение и наличие стабильной связи.

Внимание

Важно: Чтобы избежать конфликтов управления, связывайте только один пульт с одним приемником. Если вы меняете пульт или приемник, процедуру привязки необходимо повторить.

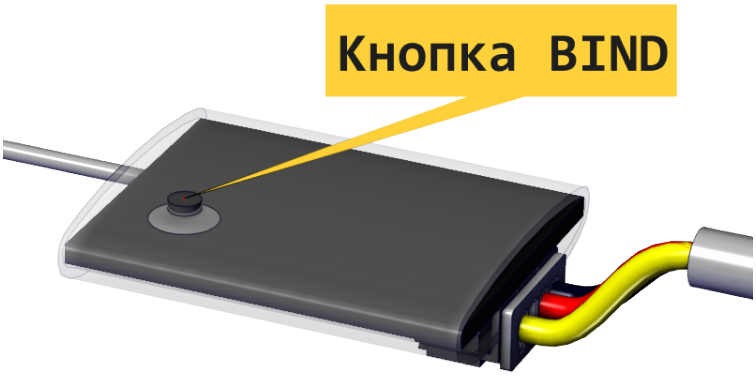


Рис. 11: Приемник FS-A8S: расположение кнопки BIND

C.2.2. Возможные проблемы и их решение

Если после выполнения процедуры квадрокоптер не реагирует на команды:

- **Проверьте протокол:** Зайдите в настройки пульта **OUTPUT MODE** и убедитесь, что выбран **PPM**. Другие протоколы (например, PWM или i-BUS) могут не восприниматься текущей прошивкой автопилота без дополнительной настройки.
- **Повторите привязку:** Возможно, вы отпустили кнопку BIND слишком рано или не успели включить режим привязки на пульте. Попробуйте снова.

- **Проверьте параметры:** Если вы летаете на Pioneer Mini, убедитесь через станцию наземного управления, что загружены корректные параметры для работы с RC-пультом.

В случае, если выполнить привязку не получается, вы всегда можете обратиться в техподдержку: support@geoscan.aero или в telegram-канал @geoscan_edu.

С.3. Стики и тумблеры

Пульт управления предоставляет пилоту гибкие возможности контроля. Основные органы управления — это два джойстика (стика) и набор переключателей (тумблеров).

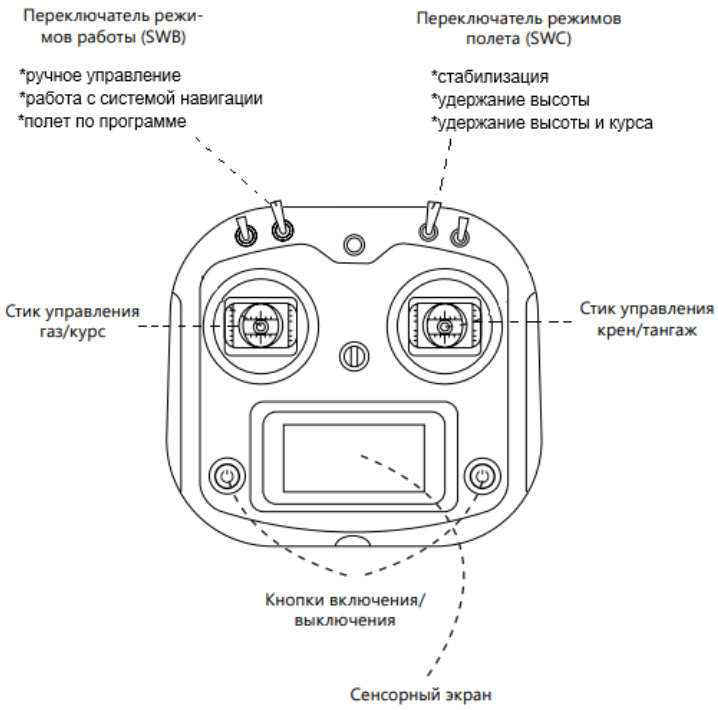


Рис. 12: Обзор органов управления пульта FS-i6S

С.3.1. Режимы управления (Тумблеры)

Тумблеры используются для переключения полетных режимов и активации функций. Ранее мы настроили тумблер **SwC** (трехпозиционный справа) на 5-й канал.

Изменение режима полета (переключатель SwC):

- **Верхнее положение (Stabilize):** Режим стабилизации. Стик газа напрямую управляет мощностью моторов. Это наиболее «ручной» режим, требующий от пилота постоянного контроля высоты. Подходит для опытных пилотов.
- **Среднее положение (AltHold):** Режим удержания высоты. Квадрокоптер использует барометр для автоматического поддержания заданной высоты. Стик газа в центре — высота не меняется. Стик вверх — набор высоты, вниз — снижение. Это рекомендуемый режим для обучения.
- **Нижнее положение (Headless):** Режим удержания высоты и курса. В дополнение к удержанию высоты, упрощается навигация. Квадрокоптер «запоминает» направление, в котором он был ориентирован при включении моторов (обычно «от пилота»). В полете, независимо от того, куда повернут нос дрона, нажатие стика «вправо» заставит его лететь вправо относительно пилота.

Внимание

Физика полета: Для определения высоты полетный контроллер использует барометр, измеряющий атмосферное давление. Помните, что резкие изменения давления (например, открытие окна в помещении или сквозняк) могут быть восприняты дроном как изменение высоты, что приведет к коррекции оборотов моторов.

Режимы работы пульта:

- **Ручное управление:** Вы полностью контролируете дрон с помощью стиков. Режим полета выбирается переключателем SwC.
- **Навигация и GPS:** В режимах с использованием системы навигации или GPS квадрокоптер управляется с пульта, но автоматически удерживает высоту и позицию, если отпустить стики.
- **Полет по программе:** Пионер выполняет загруженную миссию. Вы всегда можете переключить тумблер режимов (например, пощелкать SwC), чтобы перехватить управление и перевести дрон в ручной режим.

С.4. Управление

Пилотирование квадрокоптера требует координации и понимания того, как каждое движение стика влияет на движение аппарата в пространстве. Рассмотрим управление в режиме **Mode 2**.

С.4.1. Управление высотой (Газ / Throttle)

Левый стик, движение вверх-вниз.

- **Стик вверх:** Обороты моторов увеличиваются, подъемная сила растет, квадрокоптер набирает высоту.
- **Стик вниз:** Обороты уменьшаются, квадрокоптер снижается.

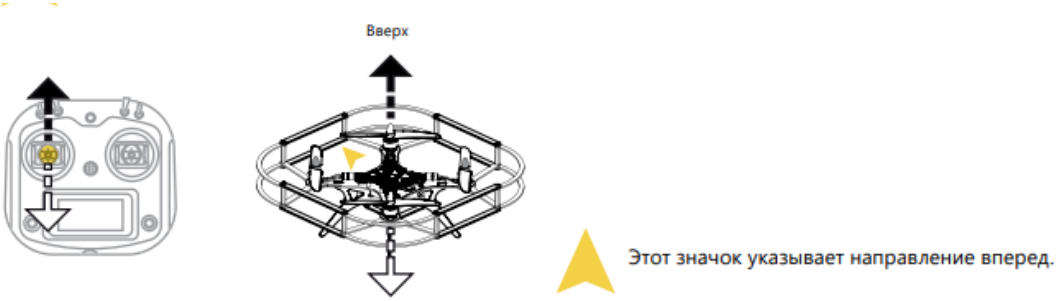


Рис. 13: Управление газом: набор высоты и снижение

Внимание

Важно: В режиме AltHold (удержание высоты) центр стика соответствует зависанию. Для взлета необходимо поднять стик выше середины.

С.4.2. Вращение по курсу (Рыскание / Yaw)

Левый стик, движение влево-вправо.

Отвечает за поворот квадрокоптера вокруг своей вертикальной оси. Это позволяет изменить направление «взгляда» дрона.

- **Стик влево:** Нос квадрокоптера поворачивается влево (против часовой стрелки).
- **Стик вправо:** Нос квадрокоптера поворачивается вправо (по часовой стрелке).

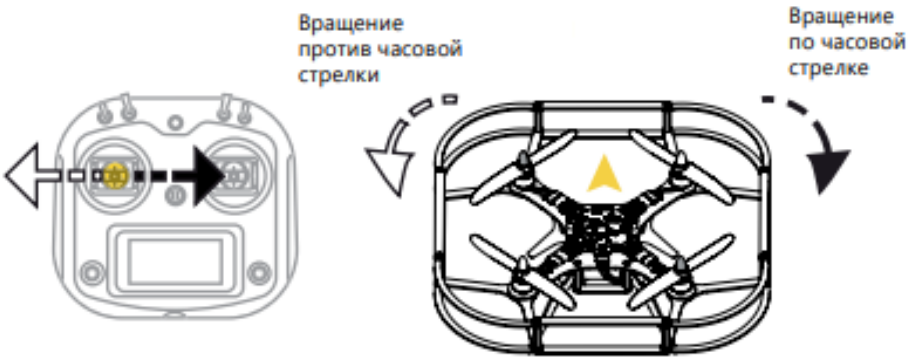


Рис. 14: Рыскание: вращение вокруг оси

C.4.3. Тангаж (Pitch) — Движение вперед/назад

Правый стик, движение вверх-вниз.
Изменяет угол наклона квадрокоптера в продольной плоскости, заставляя его двигаться горизонтально.

- **Стик вверх (от себя):** Нос опускается, дрон летит вперед.
- **Стик вниз (на себя):** Нос поднимается, дрон летит назад.

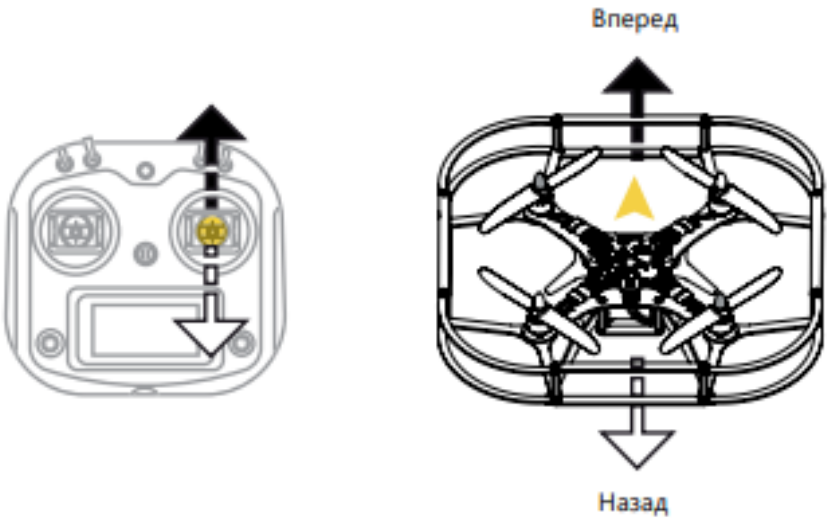


Рис. 15: Тангаж: движение вперед и назад

C.4.4. Крен (Roll) — Движение влево/вправо

Правый стик, движение влево-вправо.
Изменяет угол наклона квадрокоптера в поперечной плоскости, заставляя его смещаться вбок.

- **Стик влево:** Дрон наклоняется влево и летит боком влево.

- **Стик вправо:** Дрон наклоняется вправо и летит боком вправо.

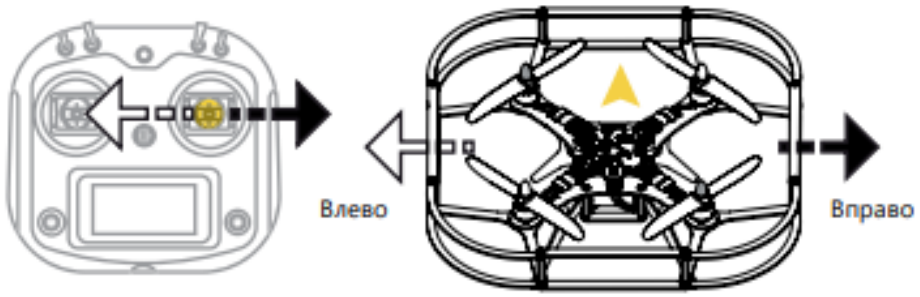


Рис. 16: Крен: движение боком

Подсказка

Совет: При обучении полетам держите хвост квадрокоптера (где установлен аккумулятор или светодиод) к себе. Так направления управления будут совпадать: стик вправо — дрон вправо. Если дрон развернется к вам носом, управление станет «зеркальным» (стик вправо — дрон влево с вашей точки зрения).

D. Методика тестирования и безопасность

D.1. Протокол безопасности полетов

D.1.1. 1. Подготовка зоны (Pre-flight)

- Очистить зону 3×3 метра.
- Закрыть окна (ветер).
- Убрать животных и детей.
- Предупредить окружающих о запуске.
- Надеть защитные очки (для оператора).

D.1.2. 2. Проверка дрона (Checklist)

- **Пропеллеры:** Целые, затянуты, вращаются свободно.
- **Аккумулятор:** Заряжен > 3.8В на банку, закреплен надежно.
- **Камера:** Линза чистая, шлейф не болтается.
- **Wi-Fi:** Сигнал стабильный (RSS > -60 dBm).

D.1.3. 3. Экстренная ситуация

Если дрон ведет себя неадекватно:

1. Громко крикнуть «ПОСАДКА!».
2. Нажать кнопку LAND (если есть).
3. Если нет реакции — аккуратно поймать снизу (за батарею) и перевернуть.
4. Отключить аккумулятор.

D.2. Тестирование в разных условиях

D.2.1. Погодные условия (Ветер)

Дрон Pioneer очень легкий (около 100 г). Он не предназначен для полетов на улице при ветре > 3 м/с. Для теста в помещении используйте вентилятор:

- Включите вентилятор на минимальную скорость с расстояния 2 метра.
- Попробуйте удержать точку (режим Optical Flow).
- Если дрон сносит > 50 см — условия непригодны для точной навигации.

D.2.2. Освещение (Lux)

Камера Pioneer имеет низкий динамический диапазон.

- **Яркое солнце:** Пересвет (белое пятно). Маркеры не видны.
- **Сумерки:** Сильный шум матрицы. Маркеры «дрожат».
- **Мерцание ламп:** Люминесцентные лампы (50 Гц) могут давать полосы на видео.

E. Готовые конфигурации (Templates)

E.1. Файл конфигурации camera_params.yaml

Те же параметры камеры, что сохраняют скрипты калибровки в `data.yml`, но в формате для сторонних инструментов: матрица — `camera_matrix`, дисторсия — `distortion_coefficients`. Примеры этой книги читают `data.yml`.

camera_params.yaml

```yaml
%YAML:1.0
image_width: 640
image_height: 480
camera_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 600.0, 0., 320.0, 0., 600.0, 240.0, 0., 0., 1. ]
distortion_coefficients: !!opencv-matrix
   rows: 1
   cols: 5
   dt: d
   data: [ 0.1, -0.2, 0., 0., 0. ]
```

E.2. Шаблон скрипта полета (main.py)

Базовый шаблон

```python
import time
from pioneer_sdk import Pioneer

def main():
    drone = Pioneer()
    try:
        drone.arm()
        time.sleep(1) # Даем автопилоту перейти в режим готовности
```

```
        drone.takeoff()
        time.sleep(3)

        # ... Ваш код ...

        drone.land()
    except Exception as e:
        print(f"Error: {e}")
        drone.land()
    finally:
        drone.disarm()
        drone.close_connection()

if __name__ == "__main__":
    main()
```

Список литературы

1. Camera calibration with OpenCV. — 2026. — URL: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html (visited on 03/04/2026).
2. MediaPipe Pose. — 2026. — URL: https://developers.google.com/mediapipe/solutions/vision/pose_landmarker (visited on 03/04/2026).
3. *Slatkin B.* Effective Python : 90 Specific Ways to Write Better Python. — 2nd ed. — Addison-Wesley Professional, 2019.