Flightmare

Yunlong Song, Selim Naji, Elia Kaufmann, Antonio Loquercio, Pro

May 15, 2023

CONTENTS

Welcome to Flightmare's documentation!

This home page contains an index with a brief description of the different sections in the documentation. Feel free to read in whatever order you prefer. In any case, here are a few suggestions for newcomers.

CHAPTER

ONE

GETTING STARTED

Introduction — What to expect from Flightmare.
Quick start — Get the latest Flightmare releases.
F.A.Q. — Some of the most frequent installation issues.

CHAPTER

TWO

BUILDING FLIGHTMARE BINARY

Standalone build — Make the build in Unity.
F.A.Q. — Some of the most frequent building issues.

CHAPTER

THREE

FIRST STEPS

Core concepts — Overview of the basic concepts in Flightmare.
Server and client — Manage and access the simulation.
Quadrotor and Objects — Learn about quadrotors and objects and how to handle them.
Environments and navigation — Discover the different maps and how to navigate through waypoints.
Sensors and data — Retrieve simulation data using sensors.
Point Cloud - Retrieve the point cloud from the scene.
F.A.Q. — Some of the most frequent usage issues.

CHAPTER

FOUR

ADVANCED STEPS

Advanced concepts — Overview of all advanced steps
Motion planning — Learn about connecting OMPL and Flightmare

CHAPTER

FIVE

TUTORIALS — GENERAL

Pilot Tutorial — Use Gazebo dynamics with Flightmare.
Racing Tutorial — Navigate through gates.
Retrieve simulation data — A step by step guide to properly gather data.

CHAPTER

SIX

TUTORIALS — ASSETS

[Todo] Add a new environment — Create and add a new environment to Flightmare.
[Todo] Add a new vehicle — Prepare a vehicle to be used in Flightmare.
[Todo] Add new objects — Import additional props into Flightmare.
[Todo] Map customization — Edit an existing map.
[Todo] Material customization — Edit vehicle and building materials.

CHAPTER

SEVEN

TUTORIALS — DEVELOPERS

[Todo] Create a sensor — Develop a new sensor to be used in Flightmare.
[Todo] Make a release — For developers who want to publish a release.
[Todo] Generate detailed colliders — Create detailed colliders for meshes.

CHAPTER

EIGHT

C++ REFERENCES

Quadrotor references — All functions of quadrotor.
Camera references — All functions of camera.
Gate references — All functions of gate.
Quadrotor Environment — All OpenAIGym environments function.

CHAPTER

NINE

PYTHON REFERENCES

Wrapper — Wrapper functions of QuadrotorEnv.

CHAPTER

TEN

CONTRIBUTING

[Todo] Contribution guidelines — The different ways to contribute to Flightmare.
[Todo] Code of conduct — Standard rights and duties for contributors.
[Todo] Coding standard — Guidelines to write proper code.
[Todo] Documentation standard — Guidelines to write proper documentation.

10.1 Flightmare documentation

Welcome to Flightmare's documentation!

This home page contains an index with a brief description of the different sections in the documentation. Feel free to read in whatever order you prefer. In any case, here are a few suggestions for newcomers.

- *Installing and getting started with Flightmare*

10.1.1 Getting started

Introduction — What to expect from Flightmare.
Quick start — Get the latest Flightmare releases.
F.A.Q. — Some of the most frequent installation issues.

10.1.2 Building Flightmare binary

Standalone build — Make the build in Unity.
F.A.Q. — Some of the most frequent building issues.

10.1.3 First steps

Core concepts — Overview of the basic concepts in Flightmare.
Server and client — Manage and access the simulation.
Quadrotor and Objects — Learn about quadrotors and objects and how to handle them.
Environments and navigation — Discover the different maps and how to navigate through waypoints.
Sensors and data — Retrieve simulation data using sensors.
Point Cloud - Retrieve the point cloud from the scene.
F.A.Q. — Some of the most frequent usage issues.

10.1.4 Advanced steps

Advanced concepts — Overview of all advanced steps
Motion planning — Learn about connecting OMPL and Flightmare

10.1.5 Tutorials — General

Pilot Tutorial — Use Gazebo dynamics with Flightmare.
Racing Tutorial — Navigate through gates.
Retrieve simulation data — A step by step guide to properly gather data.

10.1.6 Tutorials — Assets

[Todo] Add a new environment — Create and add a new environment to Flightmare.
[Todo] Add a new vehicle — Prepare a vehicle to be used in Flightmare.
[Todo] Add new objects — Import additional props into Flightmare.
[Todo] Map customization — Edit an existing map.
[Todo] Material customization — Edit vehicle and building materials.

10.1.7 Tutorials — Developers

[Todo] Create a sensor — Develop a new sensor to be used in Flightmare.
[Todo] Make a release — For developers who want to publish a release.
[Todo] Generate detailed colliders — Create detailed colliders for meshes.

10.1.8 C++ References

Quadrotor references — All functions of quadrotor.
Camera references — All functions of camera.
Gate references — All functions of gate.
Quadrotor Environment — All OpenAIGym environments function.

10.1.9 Python References

Wrapper — Wrapper functions of QuadrotorEnv.

10.1.10 Contributing

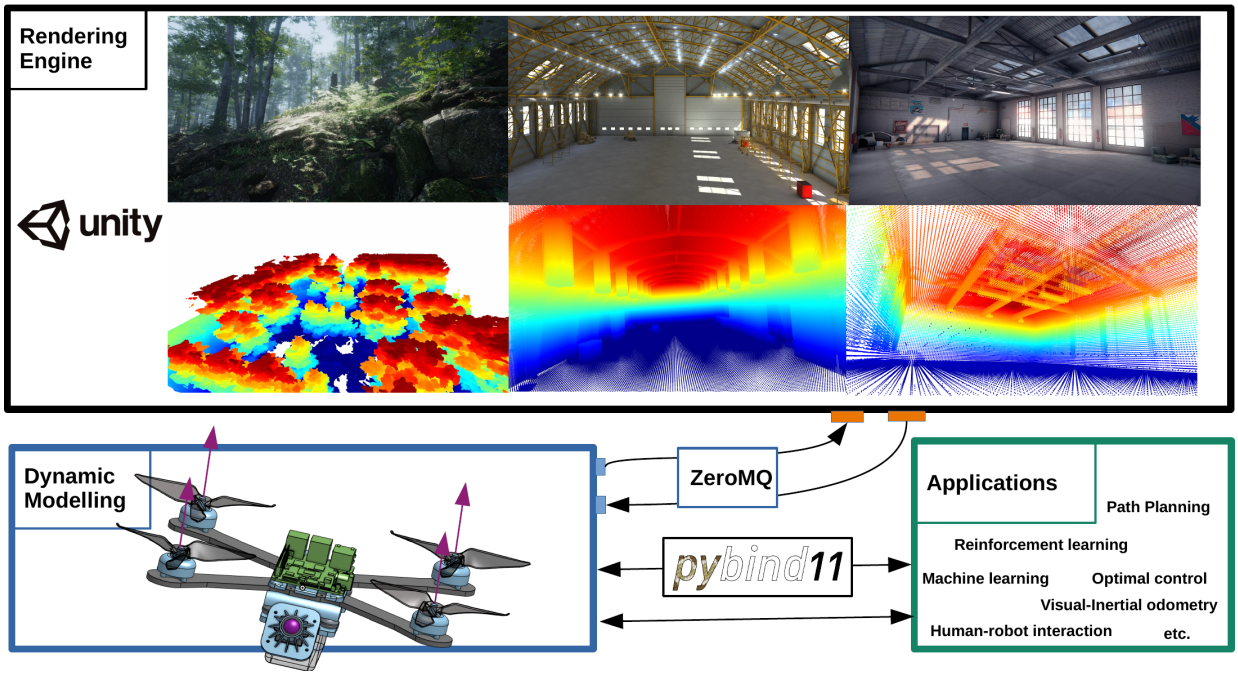
[Todo] Contribution guidelines — The different ways to contribute to Flightmare.
[Todo] Code of conduct — Standard rights and duties for contributors.
[Todo] Coding standard — Guidelines to write proper code.
[Todo] Documentation standard — Guidelines to write proper documentation.

10.2 Flightmare

Flightmare is a flexible modular quadrotor simulator. Flightmare is composed of two main components: a configurable rendering engine built on Unity and a flexible physics engine for dynamics simulation. Those two components are totally decoupled and can run independently from each other. Flightmare comes with several desirable features: (i) a large multi-modal sensor suite, including an interface to extract the 3D point-cloud of the scene; (ii) an API for reinforcement learning which can simulate hundreds of quadrotors in parallel; and (iii) an integration with a virtual-reality headset for interaction with the simulated environment. Flightmare can be used for various applications, including path-planning, reinforcement learning, visual-inertial odometry, deep learning, human-robot interaction, etc.

10.2.1 The simulator

Flightmare is an open-source simulator for quadrotors. It is composed of two main components: a configurable rendering engine built on Unity and a flexible physics engine for dynamics simulation. In addition, it has OpenAI gym-style python wrapper for reinforcement learning tasks and a flexible interface with stable baselines for solving these tasks with deep RL algorithms. Flightmare provides ROS wrapper to interface with popular ROS packages, such as high_mpc for learning-based mpc, rpg_mpc for advanced quadrotor controller, and rpg_quadrotor_control for hard-ware-in-the-loop simulation.



Software components

- flightlib: Flightmare Library
  - Quadrotor Dynamics
  - Sensors Simulation
  - Unity Bridge
  - Python Wrapper
- flightrender: Flightmare Rendering Engine
  - Photo-realistic 3D Environment
  - RGB Images, Depth, Segmentation
- flightrl: Reinforcement Learning Algorithms and Examples
  - Deep Reinforcement Learning Algorithms, e.g., PPO
  - Reinforcement learning examples, e.g., quadrotor control
- flightros: ROS Wrapper for Flightmare Library
  - ROS wrapper
  - Quadrotor Control example with PID controller, also simulate RGB Camera that can request images from Flightmare Rendering Engine.

10.2.2 Publication

Flightmare: A Flexible Quadrotor Simulator

Yunlong Song, Selim Naji, Elia Kaufmann, Antonio Loquercio, Davide Scaramuzza

Robotics and Perception Group
Depts. Informatics and Neuroinformatics
University of Zurich and ETH Zurich

Abstract: Currently available quadrotor simulators have a rigid and highly-specialized structure: either are they really fast, physically accurate, or photo-realistic. In this work, we propose a paradigm-shift in the development of simulators: moving the trade-off between accuracy and speed from the developers to the end-users. We use this design idea to develop a novel modular quadrotor simulator: Flightmare. Flightmare is composed of two main components: a configurable rendering engine built on Unity and a flexible physics engine for dynamics simulation. Those two components are totally decoupled and can run independently from each other. This makes our simulator extremely fast: rendering achieves speeds of up to 230 Hz, while physics simulation of up to 200.000 Hz. In addition, Flightmare comes with several desirable features: (i) a large multi-modal sensor suite, including an interface to extract the 3D point-cloud of the scene; (ii) an API for reinforcement learning which can simulate hundreds of quadrotors in parallel; and (iii) an integration with a virtual-reality headset for interaction with the simulated environment. We demonstrate the flexibility of Flightmare by using it for two completely different robotic tasks: learning a sensorimotor control policy for a quadrotor and path-planning in a complex 3D environment.

Website: https://uzh-rpg.github.io/flightmare
Code: https://github.com/uzh-rpg/flightmare

1 Introduction

Simulation is an invaluable tool for the robotics researcher. It allows developing and/or testing algorithms in a safe and inexpensive manner, without having to worry about the time-consuming and expensive process of dealing with real-world hardware. The ideal simulator has three main characteristics: it is (i) *fast*, to collect a large amount of data with limited time and compute; (ii) *physically-accurate*, to represent the dynamics of the real world with high-fidelity; and (iii) *photo-realistic*, to minimize the discrepancy between simulated and real-world sensors' observations. Those objectives are generally conflicting in nature: for example, the more a simulation is realistic, the slower it is. Therefore, achieving all those objectives in a single monolithic simulator is challenging.

The landscape of currently available simulators is fragmented: some are extremely fast, *e.g.* Mujoco [1], while others have either really accurate dynamics [2, 3] or highly photo-realistic rendering [4]. One of the main limitations of those simulators is their rigid nature. Specifically, they entrust the simulator developers, and not the end users, to trade-off accuracy for speed. However, this paradigm leaves some questions open: What if we want to *dynamically* change the underlying physics model? What if we want to *actively* trade-off photo-realism for speed? In this work, we answer these questions in the context of quadrotor simulation. To do so, we propose *Flightmare*, a new flexible simulator which puts the speed vs. accuracy trade-off in the hands of the end users.

Flightmare is composed of two main blocks: a rendering engine, based on Unity [5], and a physics model. These blocks are completely decoupled and can run independently from each other. In addition, each block is flexible by design. Indeed, the rendering block can be used within a wide range of 3D realistic environments and generate visual information all the way from low to high photo-realism. With minimal additional computational costs, it is also possible to simulate sensor

If you use this code in a publication, please cite the following paper PDF:

```
@article{yunlong2020flightmare,
title={Flightmare: A Flexible Quadrotor Simulator},
author={Song, Yunlong and Naji, Selim and Kaufmann, Elia and Loquercio, Antonio and␣
↪Scaramuzza, Davide},
booktitle={Conference on Robot Learning (CoRL)},
year={2020}}

}
```

10.2.3 License

This project is released under the MIT License. Please review the License file for more details.

10.2.4 Acknowledgements

This project is inspired by FlightGoggles, we use some components from FlightGoggles.

The Image Synthesis for Machine Learning from Unity is a core element of Flightmare's image post-processing.

The demo scene Industrial, which we added in the repository, was created by Dmitrii Kutsenko and is freely available in the asset store. The original asset can be found here.

10.3 Quick start

10.3.1 Prerequisites

Git

For the following installation instructions you will need git which you can set up with the following commands:

```
sudo apt-get install git
git config --global user.name "Your Name Here"
git config --global user.email "Same Email as used for github"
git config --global color.ui true
```

Packages

Flightmare requires CMake and GCC compiler. You will also need system packages python3, OpenMPI, and OpenCV.

```
apt-get update && apt-get install -y --no-install-recommends \
   build-essential \
   cmake \
   libzmqpp-dev \
   libopencv-dev
```

10.3.2 Install with pip

In this section, we assume that all the prerequisites are installed.

Python environment

It is a good idea to use virtual environments (virtualenvs) or Anaconda to make sure packages from different projects do not interfere with each other. Check here for Anaconda installation.

1. To create an environment with python3.6

```
conda create --name ENVNAME python=3.6
```

2. Activate a named Conda environment

```
conda activate ENVNAME
```

Install Flightmare

Clone the project to your desktop (or any other directory)

```
cd ~/Desktop
git clone https://github.com/uzh-rpg/flightmare.git
```

Add Environment Variable

Add **FLIGHTMARE_PATH** environment variable to your *.bashrc* file:

```
echo "export FLIGHTMARE_PATH=~/Desktop/flightmare" >> ~/.bashrc
source ~/.bashrc
```

Install dependencies

```
conda activate ENVNAME
cd flightmare/
pip install -r requirements.txt
```

Install Flighmare (flightlib)

```
cd flightmare/flightlib
# it first compile the flightlib and then install it as a python package.
pip install .
```

After installing flightlib, you can following the [[Basic Usage with Python|Basic-Usage-with-Python]] for some Reinforcement learning examples.

10.3.3 Install with ROS

In this section, we assume that all the prerequisites are installed.

Get ROS

You can use this framework with the Robot Operating System ROS and you therefore first need to install it (Desktop-Full Install) by following the steps described in the ROS Installation.

Gazebo

To install Gazebo checkout out their documentation.

Or in short * ROS Melodic and newer: use Gazebo version 9.x *sudo apt-get install gazebo9* * ROS Kinetic and newer: use Gazebo version 7.x *sudo apt-get install gazebo7* * ROS Indigo: use Gazebo version 2.x *sudo apt-get install gazebo2*

ROS Dependencies

Install system and ROS dependencies (on **Ubuntu20.04**, replace *python-vcstool* with *python3-vcstool*):

```
sudo apt-get install libgoogle-glog-dev protobuf-compiler ros-$ROS_DISTRO-octomap-msgs␣
↪ros-$ROS_DISTRO-octomap-ros ros-$ROS_DISTRO-joy python-vcstool
```

Before continuing, make sure that your protobuf compiler version is 3.0.0. To check this out, type in a terminal `protoc --version`. If This is not the case, then check out this guide on how to do it.

Get catkin tools

Get catkin tools with the following commands:

```
sudo apt-get install python-pip
sudo pip install catkin-tools
```

Create a catkin workspace

Create a catkin workspace with the following commands:

```
cd
mkdir -p catkin_ws/src
cd catkin_ws
catkin config --init --mkdirs --extend /opt/ros/$ROS_DISTRO --merge-devel --cmake-args -
↪DCMAKE_BUILD_TYPE=Release
```

Install Flightmare

Clone the repository

```
cd ~/catkin_ws/src
git clone https://github.com/uzh-rpg/flightmare.git
```

Clone dependencies:

```
vcs-import < flightmare/flightros/dependencies.yaml
```

Build:

```
catkin build
```

Add sourcing of your catkin workspace and **FLIGHTMARE_PATH** environment variable to your *.bashrc* file:

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
echo "export FLIGHTMARE_PATH=~/catkin_ws/src/flightmare" >> ~/.bashrc
source ~/.bashrc
```

10.3.4 Download Flightmare Unity Binary

Download the Flightmare Unity Binary **RPG_Flightmare.tar.xz** for rendering from the Releases and extract it into the */path/to/flightmare/flightrender*.

10.3.5 Run Flightmare

Run pip

To use unity rendering, you need first download the binary from **Releases** and extract it into the **flightrender** folder. To enable unity for visualization, double click the extracted executable file **RPG_Flightmare.x84-64** and then test a pre-trained controller.

```
conda activate ENVNAME
cd /path/to/flightmare/flightrl
pip install .
cd examples
python3 run_drone_control.py --train 0 --render 1
```

Run ROS

In this example, we show how to use the RotorS for the quadrotor dynamics modelling, rpg_quadrotor_control for model-based controller, and **Flightmare** for image rendering.

```
# The examples are by default not built.
catkin build flightros -DBUILD_SAMPLES:=ON

# Now you can run any example.
roslaunch flightros rotors_gazebo.launch
```

We hope this example can serve as a starting point for many other applications. For example, Flightmare can be used with other multirotor models that comes with RotorS such as AscTec Hummingbird, the AscTec Pelican, or the AscTec Firefly. The default controller in rpg_quadrotor_control is a PID controller. Users have the option to use more advanced controller in this framework, such as Perception-Aware Model Predictive Control.

10.4 Frequently Asked Questions

10.4.1 Cannot launch flightros examples?

Probably the examples have not been compiled yet. Run

```
catkin build flightros -DBUILD_SAMPLES:=ON
```

to properly install the examples.

10.4.2 OpenCV 4.2 compilation error?

This issue has been resolved by a user. See the solution in the issue.

10.5 Standalone build — Make the build in Unity

10.5.1 Install Unity and import project

This is a Unity Project that built with Unity Editor Unity 2020.1.10, thus, it requires a Unity Editor installed. An earlier version might also work, but you have to downgrade the project and might loss some Unity features, or maybe not.

Prerequisites

- 1) Install Unity Hub 2.0 for Linux(recommended), or Windows/MacOS.
  - To run Unity Hub, make the downloaded file executable first: *chmod +x UnityHub.AppImage*
- 2) Install Unity Editor (>= Unity 2020.1.10) via the Unity Hub.
  - Launch **Unity Hub**
  - Find **Installs**
  - Click **add** and select a **version** of unity
  - Click **NEXT**
  - Select **Linux Build Support** if you are using the Hub in a Windows or MacOS => DONE.

Get the project

```
# Clone the unity project to the local computer
git clone git@github.com:uzh-rpg/rpg_flightmare_unity.git
```

Introduction to Unity

If you're not familiar with the basics of Unity yet, then check out the Unity Tutorials.

10.5.2 Building Linux Standalone

Compiling the Unity application to Linux (or Windows/MacOS) standalone is straightforward. On your Unity Editor, click **File** -> **Build Settings** -> **Build**. But please make sure the following settings in case of unexpected errors:

- **Target platform** = Linux/Windows/MacOS
- **Scenes in Build** = {Top_Level_Scene - 0, Warehouse/Scenes/DemoScene - 1}
- **Fullscreen Mode** = Windowed
- **Color Space** = Linear

Note: Ignore the HDRP warning (*Build time will be extremely long*) when clicking **Build** button, it will take a while (not more than 2 minutes) for building the application for the first time, and become fast (around 10 seconds) after the first build.

10.5.3 Editing Unity Project

The first time you load the project, Unity will create an empty scene which we don't need, delete it. Load the **Top_Level_Scene** that has three basic **GameObjects**,

- **HD_Camera**: the main camera that renders images
- **Camera_Controller**: the main controller for loading different scenes, moving the camera, and retrieving camera pose & publishing images.
- **splash_window**: the UI interface.

Afterward, load a warehouse scene for demo. Move the **Game** window to the right so you can visualize both the 3D model on the left and the camera view on the right. Click **Play** to build and run the program. The application creates two TCP sockets, one for publishing images and one for subscribing camera pose. Clicking the **Start** button will start the program, but since we don't have a client running and sending messages to the server, you won't see any changes.

Adding objects to scene during runtime

On the Unity side, add a prefab of your object in the folder Assets/Resources/<prefab_id>.

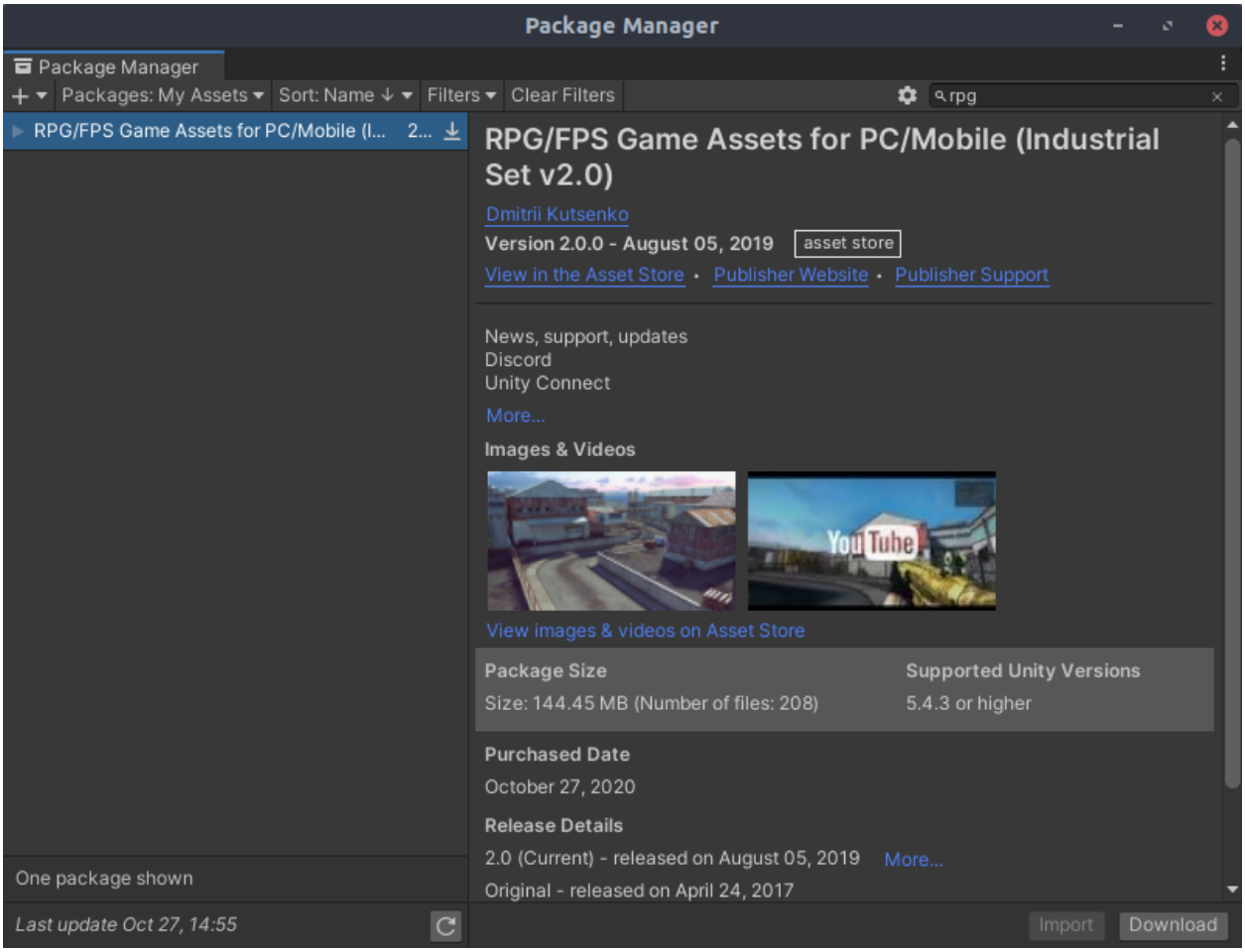
On the ROS side inherit from **static_object** in flightlib/objects. For references check out static_gate.

Add a scene to the standalone

Follow the step-by-step instructions to properly add your scene to Flightmare.

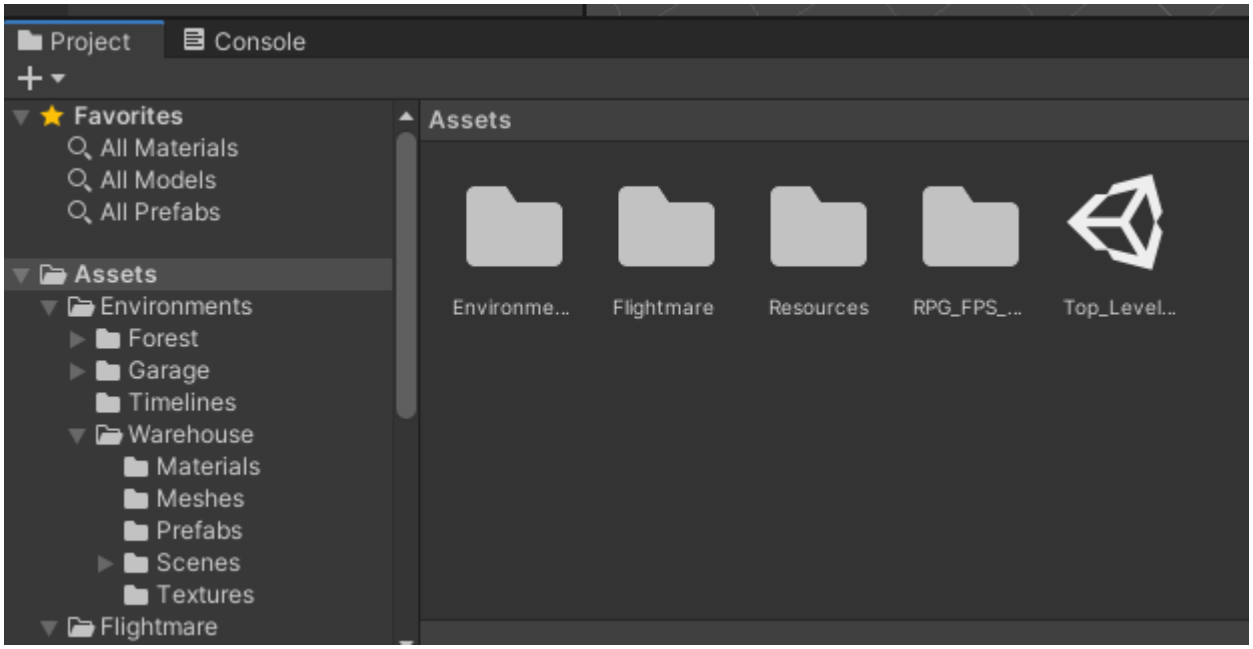
Create or add a new scene

First, you need a new scene. You can add your scenes or scenes from the asset store.



Move the scene into the folder Environments

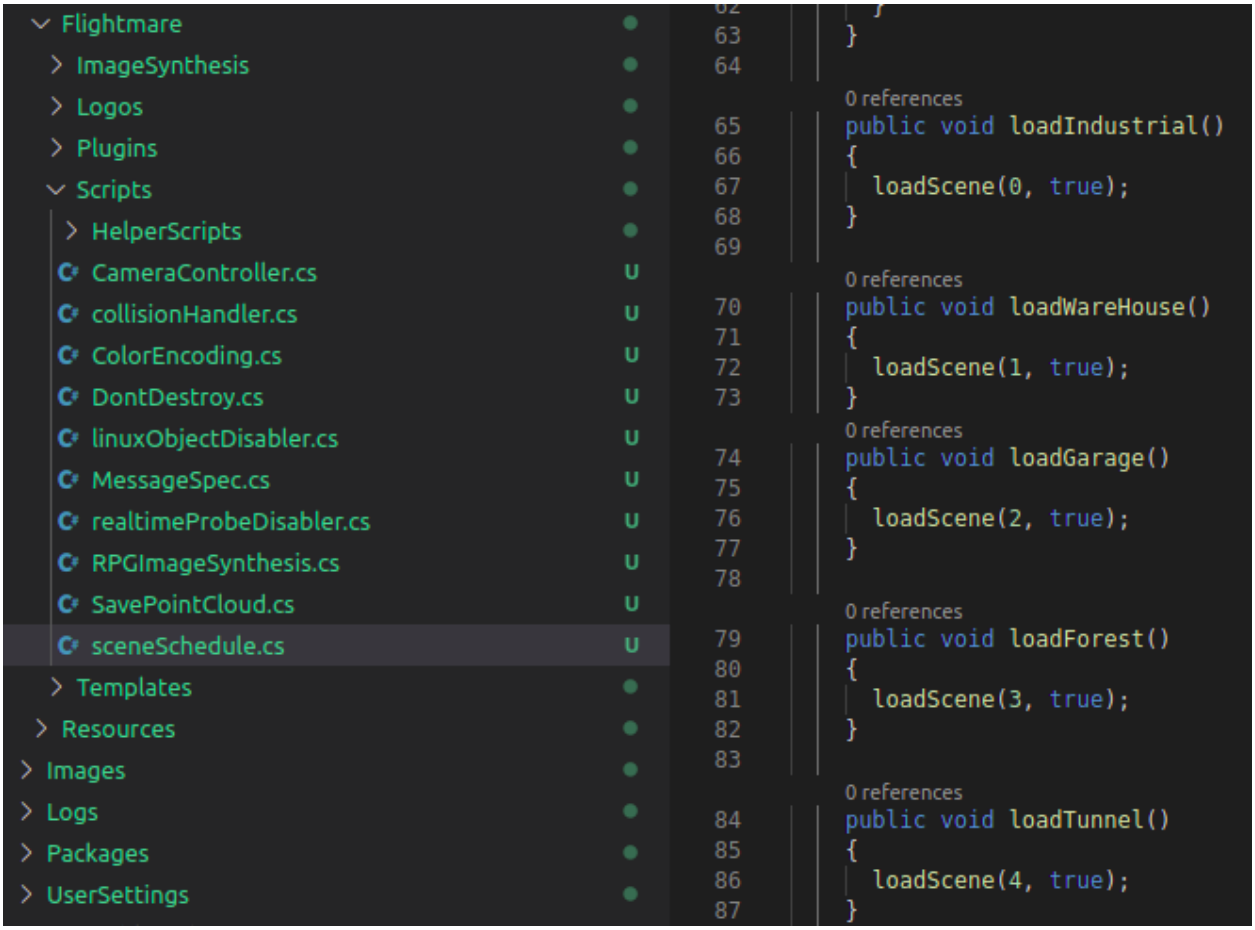
To have a consistent project structure, we move all scenes into the environment folder. The structure then is *Environments/<Project Name>/*. Within this folder, we have usually a scenes folder *Environments/<Project_Name>/Scenes/<Scene>* with all scenes. Everything else is project-depended.



Add the scene to the sceneSchedule

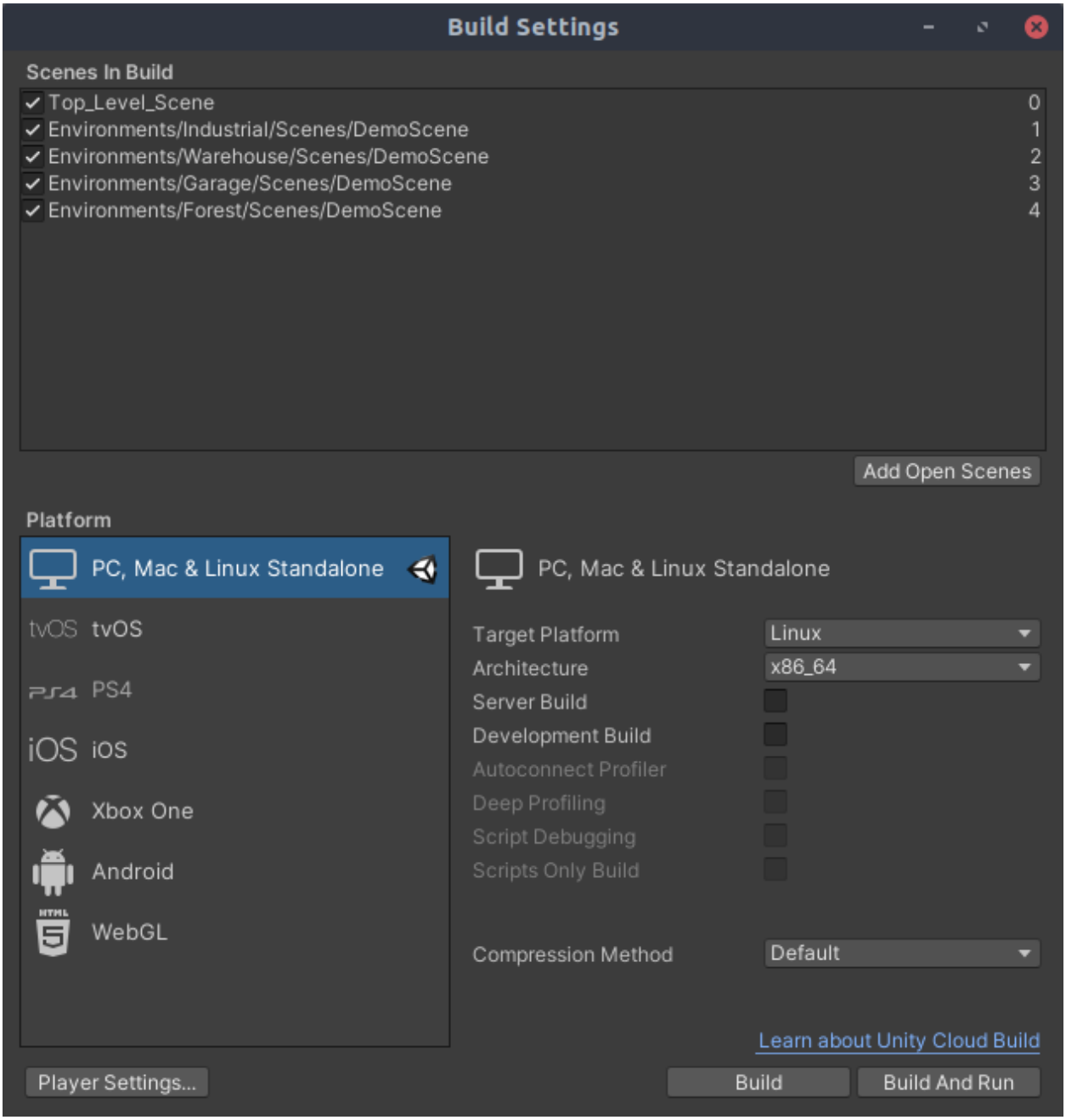
Next, we add the scene to the sceneSchedule. We have to add the path to the scene to the scenes_list. (Optional) For the UI we need to add the function load<Scene>() which will be added later to the button.

```
2 references
public class Scenes
{
    1 reference
    public const string SCENE_WAREHOUSE = "Environments/Warehouse/Scenes/DemoScene";
    1 reference
    public const string SCENE_GARAGE = "Environments/Garage/Scenes/DemoScene";
    1 reference
    public const string SCENE_TUNNELS = "Environments/Tunnels/Scenes/DemoSceneSimple";
    1 reference
    public const string SCENE_NATUREFOREST = "Environments/Forest/Scenes/DemoScene";
    1 reference
    public const string SCENE_INDUSTRIAL = "Environments/Industrial/Scenes/DemoScene";
    //
    8 references
    public List<string> scenes_list = new List<string>();
    3 references
    public int default_scene_id;
    2 references
    public int num_scene; // number of scenes in total
    1 reference
    public Scenes()
    {
      scenes_list.Add(SCENE_INDUSTRIAL);
      scenes_list.Add(SCENE_WAREHOUSE);
      scenes_list.Add(SCENE_GARAGE);
      scenes_list.Add(SCENE_NATUREFOREST);
      scenes_list.Add(SCENE_TUNNELS);

      default_scene_id = 0;
      num_scene = scenes_list.Count;
    }

}
```



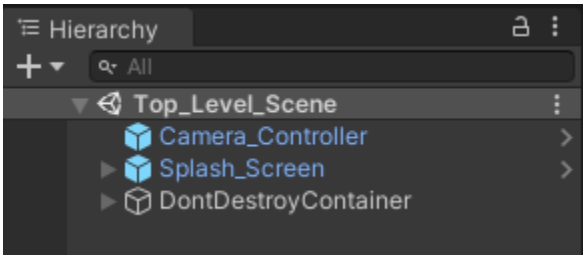
Add the scene to build settings

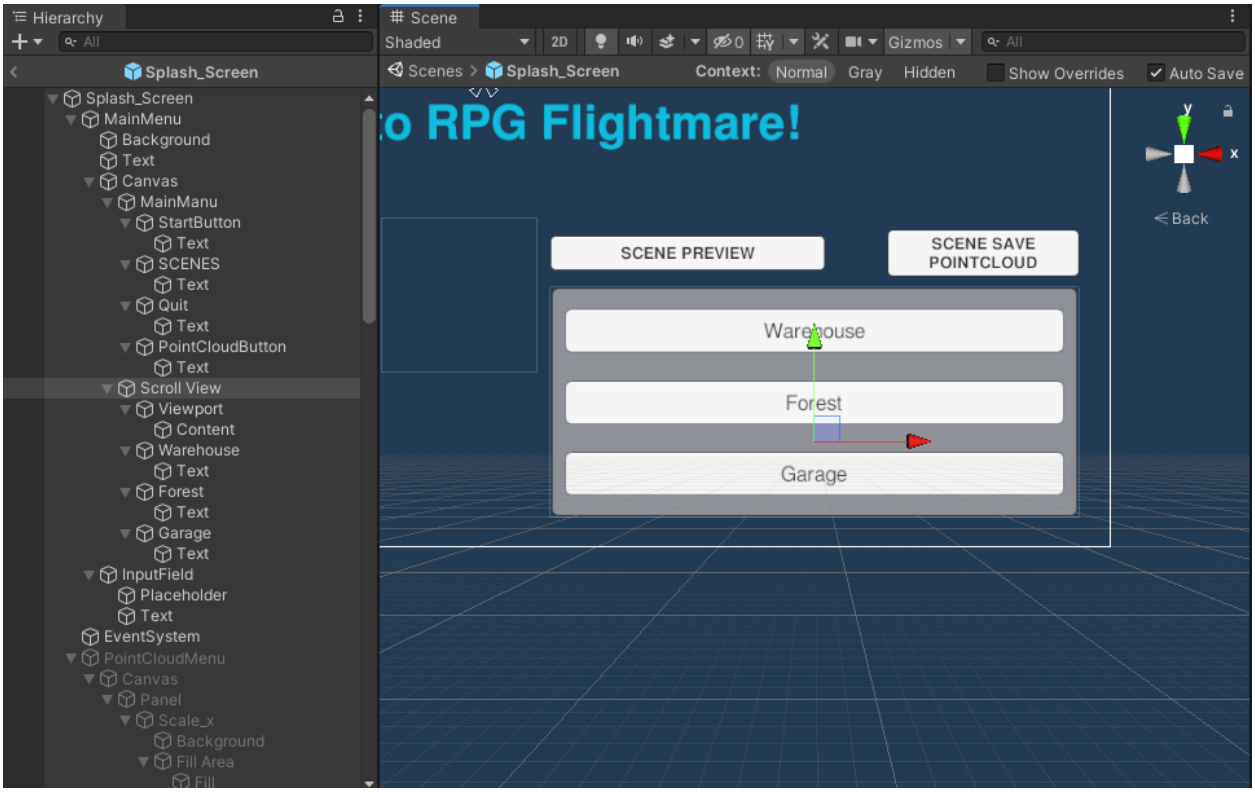
The scene also needs to be added to the build settings. Either add the currently open scene or drag and drop the scene into the build settings. The index of the scene is important.

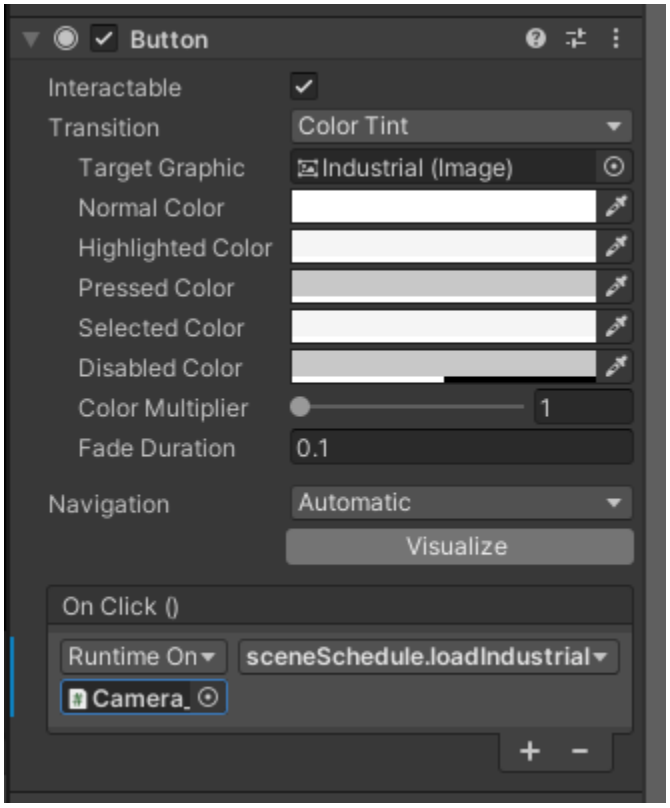

Build Settings
Scenes In Build
Top_Level_Scene 0
Environments/Industrial/Scenes/DemoScene 1
Environments/Warehouse/Scenes/DemoScene 2
Environments/Garage/Scenes/DemoScene 3
Environments/Forest/Scenes/DemoScene 4
Add Open Scenes
Platform
PC, Mac & Linux Standalone
tvOS tvOS
PS4
iOS iOS
Xbox One
Android
WebGL
PC, Mac & Linux Standalone
Target Platform Linux
Architecture x86_64
Server Build
Development Build
Autoconnect Profiler
Deep Profiling
Script Debugging
Scripts Only Build
Compression Method Default
Learn about Unity Cloud Build
Player Settings...
Build
Build And Run

(Optional) Add button on the start menu for your scene

Optionally, we modify the splash screen by adding a button in the UI. We add the Load<Scene>() to the OnClick() function of the button.







(Optional) Add a timeline for preview animation

Optionally, we add a timeline to the scene. All timelines are saved in the *Environments/timelines* folder. For the timeline, the following steps need to be done:

- Add an HDCamera to the scene from *Flightmare/Templates/Camera/HDCamera.prefab*
- Record an animation for the camera
- Add an empty GameObject to the scene and add the timeline to it
- Add an animator to the HDCamera and select the camera by the GameObject with the timeline.

Add the scene to flightlib

To be able to launch the scene from the C++ client, you will need to add the scene to flightlib in the *flightlib/include/flightlib/bridges/unity_message_types.hpp*.

```
enum UnityScene {
  INDUSTRIAL = 0,
  WAREHOUSE = 1,
  GARAGE = 2,
  NATUREFOREST = 3,
  TUNNELS = 4,
  // total number of environment
  SceneNum = 5
};
```

10.6 Frequently Asked Questions

In case your problem is not solved yet, please state it in issues.

10.6.1 How many scenes are there?

The binaries is released with three environments.

- Industrial
- Warehouse
- Garage
- NaturalForest

The git repository version only includes the environment **Industrial**, which was created by Dmitrii Kutsenko, and is freely available in the asset store.

10.6.2 Can custom scenes be added?

Yes, follow the steps *here*.

Some cool free environments:

- City
- Flooded Grounds
- Dream Forest

10.6.3 Why are some objects flickering?

Unity optimizes the performance by not render objects that are out of sight. This can lead to flickering when the camera moves quickly. This can be prevented when the option **occlusion culling** is disabled for the camera.

10.6.4 The terrain does not show in the depth image?

Unity optimizes the performance by drawing the terrain instanced. When the option **Terrain.drawInstanced is disabled** then the terrain is visible on the depth camera.

10.6.5 Why do the terrain trees only show as capsules in the pointcloud?

Unity allows (probably for performance reasons) only capsules colliders on terrain trees. We wrote a script that replaces all terrain trees with a prefab of a tree with Mesh collider on start of the scene. The script needs to be added to the specific scene.

10.6.6 Is it possible to run the standalone built on a server?

The latest release built is for local desktop environments. Technically it's possible to run it on a server, but currently there are problems with some shaders.

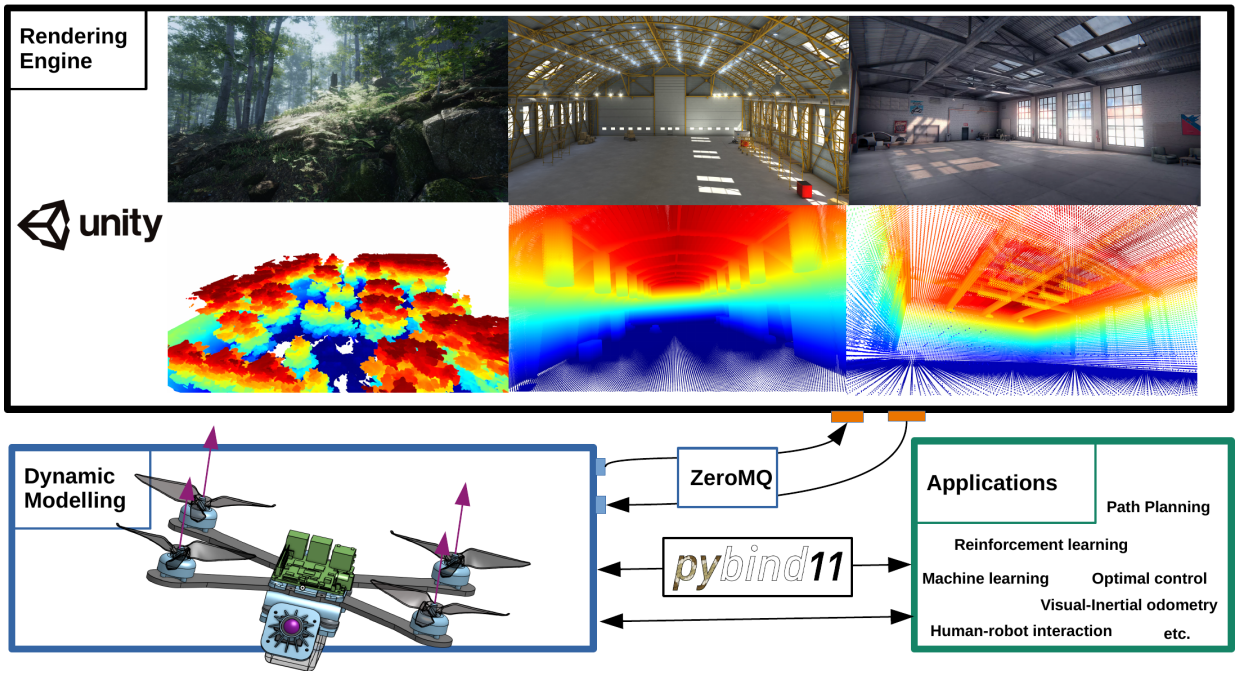
10.6.7 The depth image is not linear?

Be careful that the color space is set to **linear**. The shaders, which compute the depth image, depend on the color space. In case you change the color space in your project, you will need to adjust the UberReplacementShader. Here why we chose the factor 2.2 for our shader.

10.7 Core concepts

This page introduces the main features and modules in Flightmare. Detailed explanations of the different subjects can be found on their corresponding page.

10.7.1 Server and Client



The client is the module the user runs to model the dynamics in the simulation. A client runs with an IP and a specific port. It communicates with the server via terminal.

The server is the rendering engine representing the simulation. It contains the main methods to spawn quadrotors, change the environment, get the current state of the environment, etc.

10.7.2 Quadrotor and Objects

An actor is anything that plays a role in the simulation and is loaded dynamically into the environment.

- Quadrotor
- Sensors
- Gates

The prefabs of the actor can be found in *Assets/Resources/* on the server-side. They can be added by the client over the unity-bridge.

10.7.3 Environments and navigation

The environment is the object representing the simulated world. There are four environments available in the binary.

- Industrial
- Warehouse
- Garage
- NaturalForest

Custom environments can be added (see *here*).

Some cool free environments:

- City
- Flooded Grounds
- Dream Forest

For navigation we use the library: rpg_quadrotor_control.

10.7.4 Sensors and data

Sensors wait for rendering to happen, and then gather data from the simulation. They call for a function defining how to handle the data. Depending on which, sensors retrieve different types of sensor data.

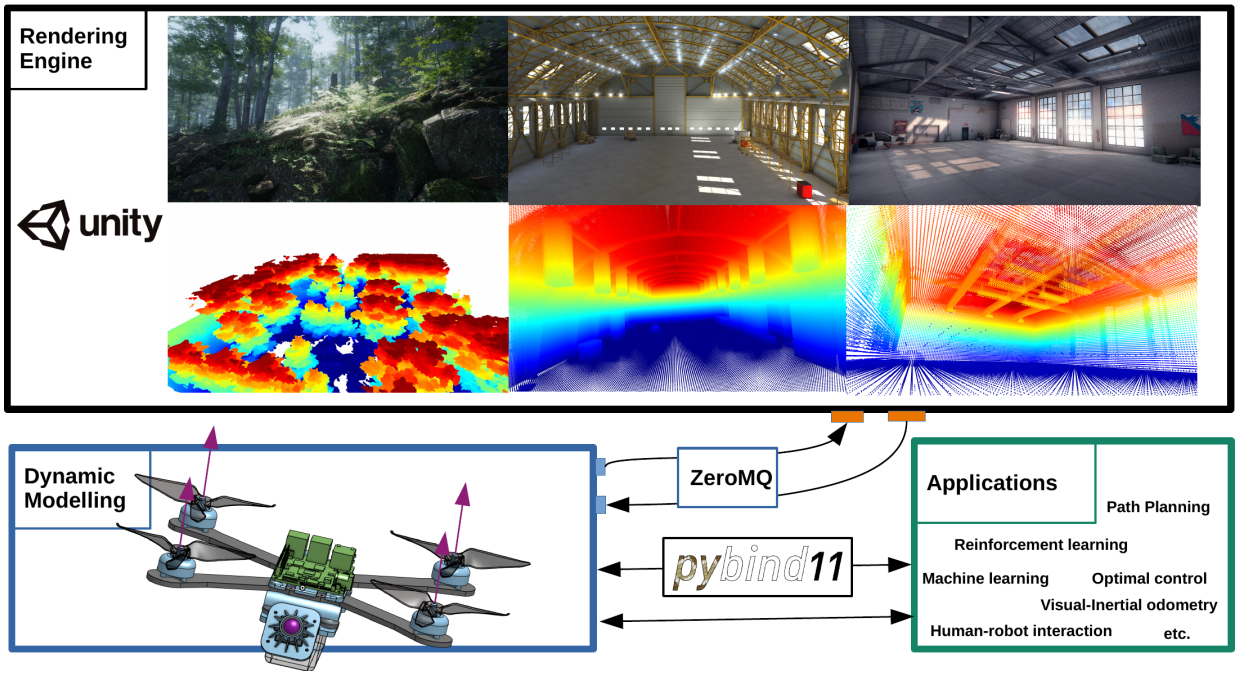
A sensor is an actor attached to a parent quadrotor. It follows the quadrotor around, gathering information about the surroundings. The following sensors are available:

- Cameras
  - RGB
  - Depth
  - Semantic segmentation
- Collision detector
- Soon to be added
  - IMU sensor
  - Lidar raycast
  - Optical flow camera
  - Event-based camera

10.7.5 Point Cloud

Flightmare can extract the point cloud of a scene. This data can be used for path planning.

10.8 Server and client



The client and the server are two of the fundamentals of Flightmare.

This tutorial goes from defining the basics and creation of these elements, to describing their possibilities.

10.8.1 The client

The client is one of the main elements in the Flightmare architecture. It connects to the server, retrieves information, and commands changes. That is done via scripts, mainly C++. There is also a minimal OpenGym Python wrapper for reinforcement learning tasks.

Only basic commands will be covered in this section. These are useful for things such as spawning lots of actors. The rest of the features are more complex, and they will be addressed in their respective pages in *Advanced steps*.

Connect to server

The server needs to be running. By default, Flightmare client broadcasts to all IP via ZMQ, and connects to the two ports 10253 and 10254 for publishing respectively subscribing. The default timeout is 10 seconds.

C++:

```
#include "flightlib/bridges/unity_bridge.hpp"

std::shared_ptr<UnityBridge> unity_bridge_ptr_ = UnityBridge::getInstance();
bool unity_ready_ = unity_bridge_ptr_->connectUnity(UnityScene::INDUSTRIAL);
```

Python:

```
#!/usr/bin/env python3
from ruamel.yaml import YAML, dump, RoundTripDumper

import os

from rpg_baselines.envs import vec_env_wrapper as wrapper
from flightgym import QuadrotorEnv_v1

# select UnityScene in flightmare/flightlib/configs/vec_env.yaml
cfg = YAML().load(open(os.environ["FLIGHTMARE_PATH"] +
                       "/flightlib/configs/vec_env.yaml", 'r'))

env = wrapper.FlightEnvVec(QuadrotorEnv_v1(dump(cfg, Dumper=RoundTripDumper), False))

env.connectUnity()
```

Add objects to the server

The client has different methods related to quadrotors and objects that allow for different functionalities.

- Add quadrotors

```
unity_bridge_ptr_->addQuadrotor(std::shared_ptr<Quadrotor> quad);
```

- Add sensors

```
unity_bridge_ptr_->addCamera(std::shared_ptr<UnityCamera> unity_camera);
```

- Add objects

```
unity_bridge_ptr_->addStaticObject(std::shared_ptr<StaticObject> static_object);
```

More detailed examples can be found on the following pages.

Rendering

C++

```
unity_bridge_ptr_->getRender(0);
unity_bridge_ptr_->handleOutput();
```

Python

```
env.stepUnity(action, send_id)
```

10.8.2 The server

The server is the rendering engine of the simulation. It runs as the binary or as the top level scene in the Unity editor in play mode. It receives messages from the client for the following components:

- Quadrotors and Objects in the simulation
- Sensors
- Environment
- Simulation settings

10.8.3 Debugging

That is a wrap on the server and client. The next step takes a closer look into quadrotors and objects to give life to the simulation. Keep reading to learn more.

10.9 Quadrotors and objects

This page introduces all dynamically spawn-able actors within Flightmare.

10.9.1 Unity Prefabs

Prefabs are already-made Unity GameObjects with animations and a series of attributes. These attributes include, among others, mesh, material, post-processing settings, and much more.

10.9.2 Quadrotors

In this section, we explain how you can spawn and move the quadrotor. Check the *references* for all functions.

Spawning

To spawn the quadrotor in a scene, you have to implement the following lines of code.

```
#include "flightlib/bridges/unity_bridge.hpp"
#include "flightlib/common/quad_state.hpp"
#include "flightlib/common/types.hpp"
#include "flightlib/objects/quadrotor.hpp"

using namespace flightlib;

// Initialize quadrotor
std::shared_ptr<Quadrotor> quad_ptr_ = std::make_shared<Quadrotor>();
QuadState quad_state_;
quad_state_.setZero();
quad_ptr_->reset(quad_state_);

// Initialize Unity bridge
std::shared_ptr<UnityBridge> unity_bridge_ptr_;
unity_bridge_ptr_ = UnityBridge::getInstance();

// Add quadrotor
unity_bridge_ptr_->addQuadrotor(quad_ptr_);
bool unity_ready_ = unity_bridge_ptr_->connectUnity(UnityScene::WAREHOUSE);
```

Set State

When the quadrotor is spawned, the pose can be updated by the following lines of code.

```
// Define new quadrotor state
quad_state_.x[QS::POSX] = (Scalar)position.x;
quad_state_.x[QS::POSY] = (Scalar)position.y;
quad_state_.x[QS::POSZ] = (Scalar)position.z;
quad_state_.x[QS::ATTW] = (Scalar)orientation.w;
quad_state_.x[QS::ATTX] = (Scalar)orientation.x;
quad_state_.x[QS::ATTY] = (Scalar)orientation.y;
quad_state_.x[QS::ATTZ] = (Scalar)orientation.z;

// Set new state
quad_ptr_->setState(quad_state_);

// Render next frame
unity_bridge_ptr_->getRender(0);
unity_bridge_ptr_->handleOutput();
```

10.9.3 Gates

Objects can be dynamically placed within a static scene. The class StaticGate inherit from the class StaticObject, so also other static objects can be added to the scene. The model loads the prefab matching the prefab_id in the folder Assets/Resources. Check the *references* for all functions.

Spawning

To spawn a gate in a scene, you have to implement the following lines of code.

```
#include <Eigen/Dense>

#include "flightlib/bridges/unity_bridge.hpp"
#include "flightlib/bridges/unity_message_types.hpp"
#include "flightlib/common/types.hpp"
#include "flightlib/objects/static_gate.hpp"

using namespace flightlib;

// Initialize gates
std::string object_id = "unity_gate"; // Unique name
std::string prefab_id = "rpg_gate"; // Name of the prefab in the Assets/Resources folder
std::shared_ptr<StaticGate> gate =
  std::make_shared<StaticGate>(object_id, prefab_id);
gate->setPosition(Eigen::Vector3f(0, 10, 2.5));
gate->setRotation(
  Quaternion(std::cos(0.5 * M_PI_2), 0.0, 0.0, std::sin(0.5 * M_PI_2)));

// Initialize Unity bridge
std::shared_ptr<UnityBridge> unity_bridge_ptr_;
unity_bridge_ptr_ = UnityBridge::getInstance();

// Add gates
unity_bridge_ptr_->addStaticObject(gate);
bool unity_ready_ = unity_bridge_ptr_->connectUnity(UnityScene::WAREHOUSE);
```

10.10 Environments and navigation

10.10.1 Flightmare environments

The following environments are available in UnityScene. The environment can easily be used with

```
// UnityScene::<SCENE_NAME>
SceneID scene_id_{UnityScene::INDUSTRIAL};
```

instead of using the specific ID.

ID	Environment	Summary
0	INDUSTRIAL	A basic outdoor industrial environment
1	WAREHOUSE	A small indoor warehouse environment
2	GARAGE	A small indoor garage environment
3	NATUREFOREST	A high-quality outdoor forest environment

10.10.2 Navigating through waypoints

In this section, we show how to use rpg_quadrotor_control with Flightmare to navigate through waypoints.

Trajectory

Generate trajectory

```
// Define path through gates
std::vector<Eigen::Vector3d> way_points;
way_points.push_back(Eigen::Vector3d(0, 10, 2.5));
way_points.push_back(Eigen::Vector3d(5, 0, 2.5));
way_points.push_back(Eigen::Vector3d(0, -10, 2.5));
way_points.push_back(Eigen::Vector3d(-5, 0, 2.5));

std::size_t num_waypoints = way_points.size();
Eigen::VectorXd segment_times(num_waypoints);
segment_times << 10.0, 10.0, 10.0, 10.0;
Eigen::VectorXd minimization_weights(5);
minimization_weights << 0.0, 1.0, 1.0, 1.0, 1.0;

polynomial_trajectories::PolynomialTrajectorySettings trajectory_settings =
  polynomial_trajectories::PolynomialTrajectorySettings(
    way_points, minimization_weights, 7, 4);

polynomial_trajectories::PolynomialTrajectory trajectory =
  polynomial_trajectories::minimum_snap_trajectories::
    generateMinimumSnapRingTrajectory(segment_times, trajectory_settings,
                                      20.0, 20.0, 6.0);
```

Get point from trajectory

```
manual_timer timer;
timer.start();

while (ros::ok()) {
  timer.stop();

  quadrotor_common::TrajectoryPoint desired_pose =
    polynomial_trajectories::getPointFromTrajectory(
      trajectory, ros::Duration(timer.get() / 1000));
```

(continues on next page)

(continued from previous page)

```
  // Set pose
  quad_state_.x[QS::POSX] = (Scalar)desired_pose.position.x();
  quad_state_.x[QS::POSY] = (Scalar)desired_pose.position.y();
  quad_state_.x[QS::POSZ] = (Scalar)desired_pose.position.z();
  quad_state_.x[QS::ATTW] = (Scalar)desired_pose.orientation.w();
  quad_state_.x[QS::ATTX] = (Scalar)desired_pose.orientation.x();
  quad_state_.x[QS::ATTY] = (Scalar)desired_pose.orientation.y();
  quad_state_.x[QS::ATTZ] = (Scalar)desired_pose.orientation.z();

  quad_ptr_->setState(quad_state_);

  unity_bridge_ptr_->getRender(0);
  unity_bridge_ptr_->handleOutput();
}
```

10.10.3 Run example

```
roslaunch flightros racing.launch
```

Here the full code example

```
#pragma once

// ros
#include <cv_bridge/cv_bridge.h>
#include <image_transport/image_transport.h>
#include <ros/ros.h>

// standard libraries
#include <assert.h>
#include <Eigen/Dense>
#include <chrono>
#include <cmath>
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <thread>
#include <vector>

// flightlib
#include "flightlib/bridges/unity_bridge.hpp"
#include "flightlib/bridges/unity_message_types.hpp"
#include "flightlib/common/quad_state.hpp"
#include "flightlib/common/types.hpp"
#include "flightlib/objects/quadrotor.hpp"
#include "flightlib/objects/static_gate.hpp"
#include "flightlib/sensors/rgb_camera.hpp"
```

(continues on next page)

(continued from previous page)

```
// trajectory
#include <polynomial_trajectories/minimum_snap_trajectories.h>
#include <polynomial_trajectories/polynomial_trajectories_common.h>
#include <polynomial_trajectories/polynomial_trajectory.h>
#include <polynomial_trajectories/polynomial_trajectory_settings.h>
#include <quadrotor_common/trajectory_point.h>

using namespace flightlib;

namespace racing {

class manual_timer {
  std::chrono::high_resolution_clock::time_point t0;
  double timestamp{0.0};

 public:
  void start() { t0 = std::chrono::high_resolution_clock::now(); }
  void stop() {
    timestamp = std::chrono::duration<double>(
                  std::chrono::high_resolution_clock::now() - t0)
                  .count() *
                1000.0;
  }
  const double &get() { return timestamp; }
};

// void setupQuad();
bool setUnity(const bool render);
bool connectUnity(void);

// unity quadrotor
std::shared_ptr<Quadrotor> quad_ptr_;
std::shared_ptr<RGBCamera> rgb_camera_;
QuadState quad_state_;

// Flightmare(Unity3D)
std::shared_ptr<UnityBridge> unity_bridge_ptr_;
SceneID scene_id_{UnityScene::WAREHOUSE};
bool unity_ready_{false};
bool unity_render_{true};
RenderMessage_t unity_output_;
uint16_t receive_id_{0};
}  // namespace racing
```

```
#include "flightros/racing/racing.hpp"

bool racing::setUnity(const bool render) {
  unity_render_ = render;
  if (unity_render_ && unity_bridge_ptr_ == nullptr) {
    // create unity bridge
    unity_bridge_ptr_ = UnityBridge::getInstance();
```

(continues on next page)

(continued from previous page)

```
    unity_bridge_ptr_->addQuadrotor(quad_ptr_);
    std::cout << "Unity Bridge is created." << std::endl;
  }
  return true;
}

bool racing::connectUnity() {
  if (!unity_render_ || unity_bridge_ptr_ == nullptr) return false;
  unity_ready_ = unity_bridge_ptr_->connectUnity(scene_id_);
  return unity_ready_;
}


int main(int argc, char *argv[]) {
  // initialize ROS
  ros::init(argc, argv, "flightmare_gates");
  ros::NodeHandle nh("");
  ros::NodeHandle pnh("~");
  ros::Rate(50.0);

  // quad initialization
  racing::quad_ptr_ = std::make_unique<Quadrotor>();
  // add camera
  racing::rgb_camera_ = std::make_unique<RGBCamera>();

  // Flightmare
  Vector<3> B_r_BC(0.0, 0.0, 0.3);
  Matrix<3, 3> R_BC = Quaternion(1.0, 0.0, 0.0, 0.0).toRotationMatrix();
  std::cout << R_BC << std::endl;
  racing::rgb_camera_->setFOV(90);
  racing::rgb_camera_->setWidth(720);
  racing::rgb_camera_->setHeight(480);
  racing::rgb_camera_->setRelPose(B_r_BC, R_BC);
  racing::rgb_camera_->setPostProcesscing(std::vector<bool>{
    false, false, false});  // depth, segmentation, optical flow
  racing::quad_ptr_->addRGBCamera(racing::rgb_camera_);

  // // initialization
  racing::quad_state_.setZero();
  racing::quad_ptr_->reset(racing::quad_state_);

  // Initialize gates
  std::string object_id = "unity_gate";
  std::string prefab_id = "rpg_gate";
  std::shared_ptr<StaticGate> gate_1 =
    std::make_shared<StaticGate>(object_id, prefab_id);
  gate_1->setPosition(Eigen::Vector3f(0, 10, 2.5));
  gate_1->setRotation(
    Quaternion(std::cos(0.5 * M_PI_2), 0.0, 0.0, std::sin(0.5 * M_PI_2)));

  std::string object_id_2 = "unity_gate_2";
  std::shared_ptr<StaticGate> gate_2 =
```

(continues on next page)

(continued from previous page)

```
  std::make_unique<StaticGate>(object_id_2, prefab_id);
gate_2->setPosition(Eigen::Vector3f(0, -10, 2.5));
gate_2->setRotation(
  Quaternion(std::cos(0.5 * M_PI_2), 0.0, 0.0, std::sin(0.5 * M_PI_2)));

// Set unity bridge
racing::setUnity(racing::unity_render_);

// Add gates
racing::unity_bridge_ptr_->addStaticObject(gate_1);
racing::unity_bridge_ptr_->addStaticObject(gate_2);

// connect unity
racing::connectUnity();

// Define path through gates
std::vector<Eigen::Vector3d> way_points;
way_points.push_back(Eigen::Vector3d(0, 10, 2.5));
way_points.push_back(Eigen::Vector3d(5, 0, 2.5));
way_points.push_back(Eigen::Vector3d(0, -10, 2.5));
way_points.push_back(Eigen::Vector3d(-5, 0, 2.5));

std::size_t num_waypoints = way_points.size();
Eigen::VectorXd segment_times(num_waypoints);
segment_times << 10.0, 10.0, 10.0, 10.0;
Eigen::VectorXd minimization_weights(5);
minimization_weights << 0.0, 1.0, 1.0, 1.0, 1.0;

polynomial_trajectories::PolynomialTrajectorySettings trajectory_settings =
  polynomial_trajectories::PolynomialTrajectorySettings(
    way_points, minimization_weights, 7, 4);

polynomial_trajectories::PolynomialTrajectory trajectory =
  polynomial_trajectories::minimum_snap_trajectories::
    generateMinimumSnapRingTrajectory(segment_times, trajectory_settings,
                                      20.0, 20.0, 6.0);

// Start racing
racing::manual_timer timer;
timer.start();

while (ros::ok() && racing::unity_render_ && racing::unity_ready_) {
  timer.stop();

  quadrotor_common::TrajectoryPoint desired_pose =
    polynomial_trajectories::getPointFromTrajectory(
      trajectory, ros::Duration(timer.get() / 1000));
  racing::quad_state_.x[QS::POSX] = (Scalar)desired_pose.position.x();
  racing::quad_state_.x[QS::POSY] = (Scalar)desired_pose.position.y();
  racing::quad_state_.x[QS::POSZ] = (Scalar)desired_pose.position.z();
  racing::quad_state_.x[QS::ATTW] = (Scalar)desired_pose.orientation.w();
  racing::quad_state_.x[QS::ATTX] = (Scalar)desired_pose.orientation.x();
```

(continues on next page)

(continued from previous page)

```
    racing::quad_state_.x[QS::ATTY] = (Scalar)desired_pose.orientation.y();
    racing::quad_state_.x[QS::ATTZ] = (Scalar)desired_pose.orientation.z();

    racing::quad_ptr_->setState(racing::quad_state_);

    racing::unity_bridge_ptr_->getRender(0);
    racing::unity_bridge_ptr_->handleOutput();
  }

  return 0;
}
```

10.11 Sensors and data

Sensors retrieve data from their surroundings. They are crucial to create a learning environment for agents.

This page summarizes everything necessary to start handling sensors. It introduces the types available and how to initialize and use them.

10.11.1 Cameras

This section explains how to spawn, listen to data of camera sensors.

It's assumed that a quadrotor has been initialized as explained on the page *Quadrotors & Objects*.

Sensor	Output	Overview
RGB	CV_8UC3	Provides clear vision of the surroundings. Looks like a normal photo of the scene.
Depth	CV_8UC3	Renders the depth of the elements in the field of view in a gray-scale map.
Segmentation	CV_8UC3	Renders elements in the field of view with a specific color according to their object type.
Optical flow	CV_8UC3	Renders the optical flow of the scene.

Check the *references* for all functions.

Spawning

This is how a camera is spawned within Flightmare. The camera is spawned at a pose relative to the quadrotor. Post-processing layers like depth, segmentation and optical flow can be enabled.

Error: The optical flow is currently incorrect

Note: Event camera in development

```
rgb_camera_ = std::make_unique<RGBCamera>();
Vector<3> B_r_BC(0.0, 0.0, 0.3);
Matrix<3, 3> R_BC = Quaternion(1.0, 0.0, 0.0, 0.0).toRotationMatrix();
rgb_camera_->setFOV(90);
rgb_camera_->setWidth(720);
rgb_camera_->setHeight(480);
rgb_camera_->setRelPose(B_r_BC, R_BC);
rgb_camera_->setPostProcesscing(
  std::vector<bool>{true, true, true});  // depth, segmentation, optical flow
quad_ptr_->addRGBCamera(rgb_camera_);
```

Listening

```
unity_bridge_ptr_->getRender(0);
unity_bridge_ptr_->handleOutput();

cv::Mat img;

rgb_camera_->getRGBImage(img); // CV_U8C3

rgb_camera_->getDepthMap(img); // CV_U8C3

rgb_camera_->getSegmentation(img); // CV_U8C3

rgb_camera_->getOpticalFlow(img); // CV_U8C3
```

Publishing

```
// initialize ROS
ros::init(argc, argv, "flightmare_rviz");
ros::NodeHandle pnh("~");
ros::Rate(50.0);

// initialize publishers
image_transport::ImageTransport it(pnh);
rgb_pub_ = it.advertise("/rgb", 1);
depth_pub_ = it.advertise("/depth", 1);
segmentation_pub_ = it.advertise("/segmentation", 1);
opticalflow_pub_ = it.advertise("/opticalflow", 1);

// ...

unity_bridge_ptr_->getRender(0);
unity_bridge_ptr_->handleOutput();

int frame_id = 0;
cv::Mat img;

rgb_camera_->getRGBImage(img);
sensor_msgs::ImagePtr rgb_msg =
```

(continues on next page)

(continued from previous page)

```
  cv_bridge::CvImage(std_msgs::Header(), "bgr8", img).toImageMsg();
rgb_msg->header.stamp.fromNSec(frame_id);
rgb_pub_.publish(rgb_msg);

rgb_camera_->getDepthMap(img);
sensor_msgs::ImagePtr depth_msg =
  cv_bridge::CvImage(std_msgs::Header(), "bgr8", img).toImageMsg();
depth_msg->header.stamp.fromNSec(frame_id);
depth_pub_.publish(depth_msg);

rgb_camera_->getSegmentation(img);
sensor_msgs::ImagePtr segmentation_msg =
  cv_bridge::CvImage(std_msgs::Header(), "bgr8", img).toImageMsg();
segmentation_msg->header.stamp.fromNSec(frame_id);
segmentation_pub_.publish(segmentation_msg);

rgb_camera_->getOpticalFlow(img);
sensor_msgs::ImagePtr opticflow_msg =
  cv_bridge::CvImage(std_msgs::Header(), "bgr8", img).toImageMsg();
opticflow_msg->header.stamp.fromNSec(frame_id);
opticalflow_pub_.publish(opticflow_msg);
```

10.11.2 Detectors

Collision

Check if your quadrotor has had a collision in the last simulation step.

Error: Not implemented yet

```
bool collision = quad_ptr_->getCollision();
```

10.12 Point Cloud

Flightmare can extract the point cloud of a scene. This data can be used for path planning.

10.12.1 Generate point cloud with flightlib

The **unity_bridge** provides a function to save the point cloud. This sends a ZMQ message from the client to the server. The server generates the point cloud and saves it at the specified file path.

```
std::shared_ptr<UnityBridge> unity_bridge_ptr_ = UnityBridge::getInstance();
bool unity_ready_ = unity_bridge_ptr_->connectUnity(UnityScene::INDUSTRIAL);

PointCloudMessage_t pointcloud_msg;
pointcloud_msg.path = "/user/";
```

(continues on next page)

(continued from previous page)

```
pointcloud_msg.file_name = "point_cloud";

// A point cloud will appear at the path /user/point_cloud.py
```

Table 1: PointCloudMessage_t

Parameter	Type	Overview
range	std::vector<Scalar>	Defines the range of the extraction area. Default (20,20,20) [m]
origin	std::vector<Scalar>	Defines the origin of the extraction area. Default (0,0,0)
resolution	Scalar	Defines the resolution of the point cloud. Default 0.15
path	std::string	The saving path of the point cloud. Default "pointcloud_data/"
file_name	std::string	The saving name of the point cloud file in .ply format. Default "default"

10.12.2 Generate point cloud manually

One way to generate the point cloud is from the start menu. Select the target scene and click on **Scene Save PointCloud**. In the now visible point cloud menu, the extraction area can be adjusted and when ready be exported by clicking on **Export PointCloud**.

The extraction area is the area of interest around the origin in a certain range. Both can be adjusted with the sliders. The extraction area is a slightly transparent red box. If not visible, you are either within the box or the box is below the ground. Further also the resolution of the point cloud can be defined. The **.ply** file can be saved at a specified file path with a specific file name. If not specified, the point cloud is saved as **default.ply** within the file path pointcloud_data/ of the Flightmare executable.

10.13 Frequently Asked Questions

10.13.1 How can we get the calibration matrix?

You don't extract the calibration matrix from Unity, instead, it is calculated on the client-side (ROS/C++).

Since both the image dimensions (width x height) and field of view (FOV) are defined by the user. (see https://github.com/uzh-rpg/flightmare/blob/master/flightlib/include/flightlib/sensors/rgb_camera.hpp#L59-L61)

you can compute the focal length using this formula: **f = (image.height / 2.0) / tan((M_PI * FOV/180.0)/2.0)** and fx=fy. Hence, the camera intrinsic matrix is **[[fx, 0, image.width/2], [0, fy, image.height/2], [0, 0, 1]]**.

10.13.2 How can we publish an image?

Follow the example in *Tutorials*.

10.13.3 Depth image rendering format results in information loss?

The issue has been mentioned by another user, but is unfortunately still unresolved. It's not straight forward to output images that are not in a 8-bit format.

10.14 Advanced concepts

In the section advanced steps, we will show how to use Flightmare in more advanced use-cases. We will continuously expand this section.

This page gives an overview of all advanced steps.

10.14.1 Motion planning

Learn about connecting The Open Motion Planning Library (OMPL) and Flightmare for advanced motion planning.

10.15 Motion planning

In this section, we explain how to use The Open Motion Planning Library (OMPL) with Flightmare for advanced motion planning.

Note: We followed the example of OMPL "Geometric Planning for a Rigid Body in 3D". Here the link to the tutorial. We highly recommend first getting familiar with OMPL before trying to integrate it into your Flightmare project.

10.15.1 OMPL Simple Setup

Setting up geometric planning for a rigid body in 3D requires the following steps:

- identify the space we are planning in: SE(3)
- select a corresponding state space from the available ones, or implement one. For SE(3), the ompl::base::SE3StateSpace is appropriate.
- since SE(3) contains an R^3 component, we need to define bounds.
- define the notion of state validity.
- define start states and a goal representation.

Once these steps are complete, the specification of the problem is conceptually done. The set of classes that allow the instantiation of this specification is shown below.

Assuming the following namespace definitions:

```
namespace ob = ompl::base;
namespace og = ompl::geometric;
```

And a state validity checking function defined like this:

```
bool isStateValid(const ob::State *state)
```

We first create an instance of the state space we are planning in.

```
void planWithSimpleSetup()
{
    // construct the state space we are planning in
    auto space(std::make_shared<ob::SE3StateSpace>());
```

We then set the bounds for the R3 component of this state space:

```
ob::RealVectorBounds bounds(3);
bounds.setLow(-1);
bounds.setHigh(1);

space->setBounds(bounds);
```

Create an instance of **ompl::geometric::SimpleSetup**. Instances of **ompl::base::SpaceInformation**, and **ompl::base::ProblemDefinition** are created internally.

```
og::SimpleSetup ss(space);
```

Set the state validity checker

```
ss.setStateValidityChecker([](const ob::State *state) { return isStateValid(state); });
```

Create a random start state:

```
ob::ScopedState<> start(space);
start.random();
```

And a random goal state:

```
ob::ScopedState<> goal(space);
goal.random();
```

Set these states as start and goal for SimpleSetup.

```
ss.setStartAndGoalStates(start, goal);
```

We can now try to solve the problem. This will also trigger a call to **ompl::geometric::SimpleSetup::setup()** and create a default instance of a planner, since we have not specified one. Furthermore, **ompl::base::Planner::setup()** is called, which in turn calls **ompl::base::SpaceInformation::setup()**. This chain of calls will lead to computation of runtime parameters such as the state validity checking resolution. This call returns a value from **ompl::base::PlannerStatus** which describes whether a solution has been found within the specified amount of time (in seconds). If this value can be cast to true, a solution was found.

```
ob::PlannerStatus solved = ss.solve(1.0);
```

If a solution has been found, we can optionally simplify it and the display it

```
if (solved)
{
    std::cout << "Found solution:" << std::endl;
    // print the path to screen
    ss.simplifySolution();
    ss.getSolutionPath().print(std::cout);
}
```

State validity checking function

A state validity function needs to be implemented. More details about how it needs to be defined can be found in the OMPL documentation. For our example, we implemented a KD-search tree and check if the point is within the range of a boundary point. If not, the point is valid and otherwise invalid.

```
bool motion_planning::isStateValid(const ob::State *state) {
// cast the abstract state type to the type we expect
const auto *se3state = state->as<ob::SE3StateSpace::StateType>();

// extract the first component of the state and cast it to what we expect
const auto *pos = se3state->as<ob::RealVectorStateSpace::StateType>(0);

// extract the second component of the state and cast it to what we expect
//  const auto *rot = se3state->as<ob::SO3StateSpace::StateType>(1);

// check validity of state defined by pos & rot
float x = pos->values[0];
float y = pos->values[1];
float z = pos->values[2];
// return a value that is always true but uses the two variables we define, so
// we avoid compiler warnings
// return isInRange(x, y, z);
Eigen::Vector3d query_pos{x, y, z};
return searchRadius(query_pos, range);
}
```

Here the helper function searchRadius.

```
bool motion_planning::searchRadius(const Eigen::Vector3d &query_point,
                                   const double radius) {
std::vector<int> indices;
std::vector<double> distances_squared;
kd_tree_.SearchRadius(query_point, radius, indices, distances_squared);

if (indices.size() == 0) {
  return true;
}

for (const auto &close_point_idx : indices) {
  // get point, check if within drone body
  Eigen::Vector3d close_point = points_.col(close_point_idx);
  // project point on each body axis and check distance
  Eigen::Vector3d close_point_body = (close_point - query_point);
  if (std::abs(close_point_body.x()) <= range &&
      std::abs(close_point_body.y()) <= range &&
      std::abs(close_point_body.z()) <= range) {
    // point is in collision
    return false;
  }
}

return true;
}
```

State space bounds

```
// set the bounds for the R^3 part of SE(3)
ob::RealVectorBounds bounds(3);

bounds.setLow(0, min_bounds.x);
bounds.setLow(1, min_bounds.y);
bounds.setLow(2, min_bounds.z);
bounds.setHigh(0, max_bounds.x);
bounds.setHigh(1, max_bounds.y);
bounds.setHigh(2, max_bounds.z);

space->setBounds(bounds);
```

Follow the OMPL documentation example so that the solution can be computed.

10.15.2 Generate Point Cloud and visualize solution path

Create a Point Cloud

Either use the GUI or the client to generate a point cloud of your environment and save it as a .ply file. Check the page *PointCloud* for more details.

For the following steps, we will assume that a point cloud was successfully saved.

Read the Point Cloud

We load the point cloud with help from tinyply.h. We populate the KD-Search Tree for more a more efficient State Validity checker function.

```
std::vector<motion_planning::float3> motion_planning::readPointCloud() {
std::unique_ptr<std::istream> file_stream;
std::vector<uint8_t> byte_buffer;
std::string filepath =
  std::experimental::filesystem::path(__FILE__).parent_path().string() +
  "/data/point_cloud.ply";
try {
  file_stream.reset(new std::ifstream(filepath, std::ios::binary));

  if (!file_stream || file_stream->fail())
    throw std::runtime_error("file_stream failed to open " + filepath);

  file_stream->seekg(0, std::ios::end);
  const float size_mb = file_stream->tellg() * float(1e-6);
  file_stream->seekg(0, std::ios::beg);

  PlyFile file;
  file.parse_header(*file_stream);

  std::cout << "\t[ply_header] Type: "
            << (file.is_binary_file() ? "binary" : "ascii") << std::endl;
  for (const auto &c : file.get_comments())
```

(continues on next page)

(continued from previous page)

```
  std::cout << "\t[ply_header] Comment: " << c << std::endl;
for (const auto &c : file.get_info())
  std::cout << "\t[ply_header] Info: " << c << std::endl;

for (const auto &e : file.get_elements()) {
  std::cout << "\t[ply_header] element: " << e.name << " (" << e.size << ")"
            << std::endl;
  for (const auto &p : e.properties) {
    std::cout << "\t[ply_header] \tproperty: " << p.name
              << " (type=" << tinyply::PropertyTable[p.propertyType].str
              << ")";
    if (p.isList)
      std::cout << " (list_type=" << tinyply::PropertyTable[p.listType].str
                << ")";
    std::cout << std::endl;
  }
}

// Because most people have their own mesh types, tinyply treats parsed data
// as structured/typed byte buffers. See examples below on how to marry your
// own application-specific data structures with this one.
std::shared_ptr<PlyData> vertices, normals, colors, texcoords, faces,
  tripstrip;

// The header information can be used to programmatically extract properties
// on elements known to exist in the header prior to reading the data. For
// brevity of this sample, properties like vertex position are hard-coded:
try {
  vertices =
    file.request_properties_from_element("vertex", {"x", "y", "z"});
} catch (const std::exception &e) {
  std::cerr << "tinyply exception: " << e.what() << std::endl;
}

manual_timer read_timer;

read_timer.start();
file.read(*file_stream);
read_timer.stop();

const float parsing_time = read_timer.get() / 1000.f;
std::cout << "\tparsing " << size_mb << "mb in " << parsing_time
          << " seconds [" << (size_mb / parsing_time) << " MBps]"
          << std::endl;

if (vertices)
  std::cout << "\tRead " << vertices->count << " total vertices "
            << std::endl;


const size_t numVerticesBytes = vertices->buffer.size_bytes();
std::vector<float3> verts(vertices->count);
```

(continues on next page)

(continued from previous page)

```
  std::memcpy(verts.data(), vertices->buffer.get(), numVerticesBytes);

  int idx = 0;
  for (const auto &point_tinyply : verts) {
    if (idx == 0) {
      points_ = Eigen::Vector3d(static_cast<double>(point_tinyply.x),
                                static_cast<double>(point_tinyply.y),
                                static_cast<double>(point_tinyply.z));
    } else {
      points_.conservativeResize(points_.rows(), points_.cols() + 1);
      points_.col(points_.cols() - 1) =
        Eigen::Vector3d(static_cast<double>(point_tinyply.x),
                        static_cast<double>(point_tinyply.y),
                        static_cast<double>(point_tinyply.z));
    }
    idx += 1;
  }
  kd_tree_.SetMatrixData(points_);

  return verts;
} catch (const std::exception &e) {
  std::cerr << "Caught tinyply exception: " << e.what() << std::endl;
}
return verts;
}
```

Execute path

In the plan function, we solve the path planning problem and can then get the found states of the solution path. In our example, we just linearly interpolated between the states and followed those points, but you can also use the rpg_quadrotor_control library.

```
path_ = ss.getSolutionPath().getStates();

for (auto const &pos : path_) {
  vecs_.push_back(stateToEigen(pos));
}
```

10.15.3 Run example

```
roslaunch flightros motion_planning.launch
```

10.15.4 Here the full code example

```
/*
 * tinyply 2.3.3 (https://github.com/ddiakopoulos/tinyply)
 *
 * A single-header, zero-dependency (except the C++ STL) public domain
 * implementation of the PLY mesh file format. Requires C++11; errors are
 * handled through exceptions.
 *
 * This software is in the public domain. Where that dedication is not
 * recognized, you are granted a perpetual, irrevocable license to copy,
 * distribute, and modify this file as you see fit.
 *
 * Authored by Dimitri Diakopoulos (http://www.dimitridiakopoulos.com)
 *
 * tinyply.h may be included in many files, however in a single compiled file,
 * the implementation must be created with the following defined prior to header
 * inclusion #define TINYPLY_IMPLEMENTATION
 *
 */

//////////////////////
//   tinyply header   //
//////////////////////

#ifndef tinyply_h
#define tinyply_h

#include <stdint.h>
#include <algorithm>
#include <cstddef>
#include <functional>
#include <map>
#include <memory>
#include <sstream>
#include <string>
#include <unordered_map>
#include <vector>

namespace tinyply {

enum class Type : uint8_t {
  INVALID,
  INT8,
  UINT8,
  INT16,
  UINT16,
```

(continues on next page)

(continued from previous page)

```
  INT32,
  UINT32,
  FLOAT32,
  FLOAT64
};

struct PropertyInfo {
  PropertyInfo(){};
  PropertyInfo(int stride, std::string str) : stride(stride), str(str) {}
  int stride{0};
  std::string str;
};

static std::map<Type, PropertyInfo> PropertyTable{
  {Type::INT8, PropertyInfo(1, std::string("char"))},
  {Type::UINT8, PropertyInfo(1, std::string("uchar"))},
  {Type::INT16, PropertyInfo(2, std::string("short"))},
  {Type::UINT16, PropertyInfo(2, std::string("ushort"))},
  {Type::INT32, PropertyInfo(4, std::string("int"))},
  {Type::UINT32, PropertyInfo(4, std::string("uint"))},
  {Type::FLOAT32, PropertyInfo(4, std::string("float"))},
  {Type::FLOAT64, PropertyInfo(8, std::string("double"))},
  {Type::INVALID, PropertyInfo(0, std::string("INVALID"))}};

class Buffer {
  uint8_t *alias{nullptr};
  struct delete_array {
    void operator()(uint8_t *p) { delete[] p; }
  };
  std::unique_ptr<uint8_t, decltype(Buffer::delete_array())> data;
  size_t size{0};

 public:
  Buffer(){};
  Buffer(const size_t size)
    : data(new uint8_t[size], delete_array()), size(size) {
    alias = data.get();
  }  // allocating
  Buffer(const uint8_t *ptr)
    : alias(const_cast<uint8_t *>(ptr)) {}  // non-allocating, todo: set size?
  uint8_t *get() { return alias; }
  const uint8_t *get_const() { return const_cast<const uint8_t *>(alias); }
  size_t size_bytes() const { return size; }
};

struct PlyData {
  Type t;
  Buffer buffer;
  size_t count{0};
  bool isList{false};
};
```

(continues on next page)

(continued from previous page)

```
struct PlyProperty {
  PlyProperty(std::istream &is);
  PlyProperty(Type type, std::string &_name)
    : name(_name), propertyType(type) {}
  PlyProperty(Type list_type, Type prop_type, std::string &_name,
              size_t list_count)
    : name(_name),
      propertyType(prop_type),
      isList(true),
      listType(list_type),
      listCount(list_count) {}
  std::string name;
  Type propertyType{Type::INVALID};
  bool isList{false};
  Type listType{Type::INVALID};
  size_t listCount{0};
};

struct PlyElement {
  PlyElement(std::istream &istream);
  PlyElement(const std::string &_name, size_t count)
    : name(_name), size(count) {}
  std::string name;
  size_t size{0};
  std::vector<PlyProperty> properties;
};

struct PlyFile {
  struct PlyFileImpl;
  std::unique_ptr<PlyFileImpl> impl;

  PlyFile();
  ~PlyFile();

  /*
   * The ply format requires an ascii header. This can be used to determine at
   * runtime which properties or elements exist in the file. Limited validation
   * of the header is performed; it is assumed the header correctly reflects the
   * contents of the payload. This function may throw. Returns true on success,
   * false on failure.
   */
  bool parse_header(std::istream &is);

  /*
   * Execute a read operation. Data must be requested via
   * `request_properties_from_element(...)` prior to calling this function.
   */
  void read(std::istream &is);

  /*
   * `write` performs no validation and assumes that the data passed into
   * `add_properties_to_element` is well-formed.
```

(continues on next page)

(continued from previous page)

```
   */
  void write(std::ostream &os, bool isBinary);

  /*
   * These functions are valid after a call to `parse_header(...)`. In the case
   * of writing, get_comments() reference may also be used to add new comments
   * to the ply header.
   */
  std::vector<PlyElement> get_elements() const;
  std::vector<std::string> get_info() const;
  std::vector<std::string> &get_comments();
  bool is_binary_file() const;

  /*
   * In the general case where |list_size_hint| is zero, `read` performs a
   * two-pass parse to support variable length lists. The most general use of
   * the ply format is storing triangle meshes. When this fact is known
   * a-priori, we can pass an expected list length that will apply to this
   * element. Doing so results in an up-front memory allocation and a
   * single-pass import, a 2x performance optimization.
   */
  std::shared_ptr<PlyData> request_properties_from_element(
    const std::string &elementKey, const std::vector<std::string> propertyKeys,
    const uint32_t list_size_hint = 0);

  void add_properties_to_element(const std::string &elementKey,
                                 const std::vector<std::string> propertyKeys,
                                 const Type type, const size_t count,
                                 const uint8_t *data, const Type listType,
                                 const size_t listCount);
};

}  // end namespace tinyply

#endif  // end tinyply_h

////////////////////////////////
//   tinyply implementation   //
////////////////////////////////

#ifdef TINYPLY_IMPLEMENTATION

#include <algorithm>
#include <cstring>
#include <functional>
#include <iostream>
#include <type_traits>

using namespace tinyply;
using namespace std;

template<typename T, typename T2>
```

(continues on next page)

(continued from previous page)

```
inline T2 endian_swap(const T &v) noexcept {
  return v;
}
template<>
inline uint16_t endian_swap<uint16_t, uint16_t>(const uint16_t &v) noexcept {
  return (v << 8) | (v >> 8);
}
template<>
inline uint32_t endian_swap<uint32_t, uint32_t>(const uint32_t &v) noexcept {
  return (v << 24) | ((v << 8) & 0x00ff0000) | ((v >> 8) & 0x0000ff00) |
         (v >> 24);
}
template<>
inline uint64_t endian_swap<uint64_t, uint64_t>(const uint64_t &v) noexcept {
  return (
    ((v & 0x00000000000000ffLL) << 56) | ((v & 0x000000000000ff00LL) << 40) |
    ((v & 0x0000000000ff0000LL) << 24) | ((v & 0x00000000ff000000LL) << 8) |
    ((v & 0x000000ff00000000LL) >> 8) | ((v & 0x0000ff0000000000LL) >> 24) |
    ((v & 0x00ff000000000000LL) >> 40) | ((v & 0xff00000000000000LL) >> 56));
}
template<>
inline int16_t endian_swap<int16_t, int16_t>(const int16_t &v) noexcept {
  uint16_t r = endian_swap<uint16_t, uint16_t>(*(uint16_t *)&v);
  return *(int16_t *)&r;
}
template<>
inline int32_t endian_swap<int32_t, int32_t>(const int32_t &v) noexcept {
  uint32_t r = endian_swap<uint32_t, uint32_t>(*(uint32_t *)&v);
  return *(int32_t *)&r;
}
template<>
inline int64_t endian_swap<int64_t, int64_t>(const int64_t &v) noexcept {
  uint64_t r = endian_swap<uint64_t, uint64_t>(*(uint64_t *)&v);
  return *(int64_t *)&r;
}
template<>
inline float endian_swap<uint32_t, float>(const uint32_t &v) noexcept {
  union {
    float f;
    uint32_t i;
  };
  i = endian_swap<uint32_t, uint32_t>(v);
  return f;
}
template<>
inline double endian_swap<uint64_t, double>(const uint64_t &v) noexcept {
  union {
    double d;
    uint64_t i;
  };
  i = endian_swap<uint64_t, uint64_t>(v);
  return d;
```

(continues on next page)

(continued from previous page)

```
}

inline uint32_t hash_fnv1a(const std::string &str) noexcept {
  static const uint32_t fnv1aBase32 = 0x811C9DC5u;
  static const uint32_t fnv1aPrime32 = 0x01000193u;
  uint32_t result = fnv1aBase32;
  for (auto &c : str) {
    result ^= static_cast<uint32_t>(c);
    result *= fnv1aPrime32;
  }
  return result;
}

inline Type property_type_from_string(const std::string &t) noexcept {
  if (t == "int8" || t == "char")
    return Type::INT8;
  else if (t == "uint8" || t == "uchar")
    return Type::UINT8;
  else if (t == "int16" || t == "short")
    return Type::INT16;
  else if (t == "uint16" || t == "ushort")
    return Type::UINT16;
  else if (t == "int32" || t == "int")
    return Type::INT32;
  else if (t == "uint32" || t == "uint")
    return Type::UINT32;
  else if (t == "float32" || t == "float")
    return Type::FLOAT32;
  else if (t == "float64" || t == "double")
    return Type::FLOAT64;
  return Type::INVALID;
}

struct PlyFile::PlyFileImpl {
  struct PlyDataCursor {
    size_t byteOffset{0};
    size_t totalSizeBytes{0};
  };

  struct ParsingHelper {
    std::shared_ptr<PlyData> data;
    std::shared_ptr<PlyDataCursor> cursor;
    uint32_t list_size_hint;
  };

  struct PropertyLookup {
    ParsingHelper *helper{nullptr};
    bool skip{false};
    size_t prop_stride{0};  // precomputed
    size_t list_stride{0};  // precomputed
  };
```

(continues on next page)

(continued from previous page)

```
  std::unordered_map<uint32_t, ParsingHelper> userData;

  bool isBinary = false;
  bool isBigEndian = false;
  std::vector<PlyElement> elements;
  std::vector<std::string> comments;
  std::vector<std::string> objInfo;
  uint8_t scratch[64];  // large enough for max list size

  void read(std::istream &is);
  void write(std::ostream &os, bool isBinary);

  std::shared_ptr<PlyData> request_properties_from_element(
    const std::string &elementKey, const std::vector<std::string> propertyKeys,
    const uint32_t list_size_hint);

  void add_properties_to_element(const std::string &elementKey,
                                 const std::vector<std::string> propertyKeys,
                                 const Type type, const size_t count,
                                 const uint8_t *data, const Type listType,
                                 const size_t listCount);

  size_t read_property_binary(const size_t &stride, void *dest,
                              size_t &destOffset, std::istream &is) noexcept;
  size_t read_property_ascii(const Type &t, const size_t &stride, void *dest,
                             size_t &destOffset, std::istream &is);

  std::vector<std::vector<PropertyLookup>> make_property_lookup_table();

  bool parse_header(std::istream &is);
  void parse_data(std::istream &is, bool firstPass);
  void read_header_format(std::istream &is);
  void read_header_element(std::istream &is);
  void read_header_property(std::istream &is);
  void read_header_text(std::string line, std::vector<std::string> &place,
                        int erase = 0);

  void write_header(std::ostream &os) noexcept;
  void write_ascii_internal(std::ostream &os) noexcept;
  void write_binary_internal(std::ostream &os) noexcept;
  void write_property_ascii(Type t, std::ostream &os, const uint8_t *src,
                            size_t &srcOffset);
  void write_property_binary(std::ostream &os, const uint8_t *src,
                             size_t &srcOffset, const size_t &stride) noexcept;
};

PlyProperty::PlyProperty(std::istream &is) : isList(false) {
  std::string type;
  is >> type;
  if (type == "list") {
    std::string countType;
    is >> countType >> type;
```

(continues on next page)

(continued from previous page)

```
    listType = property_type_from_string(countType);
    isList = true;
  }
  propertyType = property_type_from_string(type);
  is >> name;
}

PlyElement::PlyElement(std::istream &is) { is >> name >> size; }

template<typename T>
inline T ply_read_ascii(std::istream &is) {
  T data;
  is >> data;
  return data;
}

template<typename T, typename T2>
inline void endian_swap_buffer(uint8_t *data_ptr, const size_t num_bytes,
                               const size_t stride) {
  for (size_t count = 0; count < num_bytes; count += stride) {
    *(reinterpret_cast<T2 *>(data_ptr)) =
      endian_swap<T, T2>(*(reinterpret_cast<const T *>(data_ptr)));
    data_ptr += stride;
  }
}

template<typename T>
void ply_cast_ascii(void *dest, std::istream &is) {
  *(static_cast<T *>(dest)) = ply_read_ascii<T>(is);
}

int64_t find_element(const std::string &key,
                     const std::vector<PlyElement> &list) {
  for (size_t i = 0; i < list.size(); i++)
    if (list[i].name == key) return i;
  return -1;
}

int64_t find_property(const std::string &key,
                      const std::vector<PlyProperty> &list) {
  for (size_t i = 0; i < list.size(); ++i)
    if (list[i].name == key) return i;
  return -1;
}

// The `userData` table is an easy data structure for capturing what data the
// user would like out of the ply file, but an inner-loop hash lookup is
// non-ideal. The property lookup table flattens the table down into a 2D array
// optimized for parsing. The first index is the element, and the second index
// is the property.
std::vector<std::vector<PlyFile::PlyFileImpl::PropertyLookup>>
PlyFile::PlyFileImpl::make_property_lookup_table() {
```

(continues on next page)

(continued from previous page)

```
  std::vector<std::vector<PropertyLookup>> element_property_lookup;

  for (auto &element : elements) {
    std::vector<PropertyLookup> lookups;

    for (auto &property : element.properties) {
      PropertyLookup f;

      auto cursorIt = userData.find(hash_fnv1a(element.name + property.name));
      if (cursorIt != userData.end())
        f.helper = &cursorIt->second;
      else
        f.skip = true;

      f.prop_stride = PropertyTable[property.propertyType].stride;
      if (property.isList)
        f.list_stride = PropertyTable[property.listType].stride;

      lookups.push_back(f);
    }

    element_property_lookup.push_back(lookups);
  }

  return element_property_lookup;
}

bool PlyFile::PlyFileImpl::parse_header(std::istream &is) {
  std::string line;
  bool success = true;
  while (std::getline(is, line)) {
    std::istringstream ls(line);
    std::string token;
    ls >> token;
    if (token == "ply" || token == "PLY" || token == "")
      continue;
    else if (token == "comment")
      read_header_text(line, comments, 8);
    else if (token == "format")
      read_header_format(ls);
    else if (token == "element")
      read_header_element(ls);
    else if (token == "property")
      read_header_property(ls);
    else if (token == "obj_info")
      read_header_text(line, objInfo, 9);
    else if (token == "end_header")
      break;
    else
      success = false;  // unexpected header field
  }
  return success;
```

(continues on next page)

(continued from previous page)

```
}

void PlyFile::PlyFileImpl::read_header_text(std::string line,
                                            std::vector<std::string> &place,
                                            int erase) {
  place.push_back((erase > 0) ? line.erase(0, erase) : line);
}

void PlyFile::PlyFileImpl::read_header_format(std::istream &is) {
  std::string s;
  (is >> s);
  if (s == "binary_little_endian")
    isBinary = true;
  else if (s == "binary_big_endian")
    isBinary = isBigEndian = true;
}

void PlyFile::PlyFileImpl::read_header_element(std::istream &is) {
  elements.emplace_back(is);
}

void PlyFile::PlyFileImpl::read_header_property(std::istream &is) {
  if (!elements.size())
    throw std::runtime_error("no elements defined; file is malformed");
  elements.back().properties.emplace_back(is);
}

size_t PlyFile::PlyFileImpl::read_property_binary(const size_t &stride,
                                                  void *dest,
                                                  size_t &destOffset,
                                                  std::istream &is) noexcept {
  destOffset += stride;
  is.read((char *)dest, stride);
  return stride;
}

size_t PlyFile::PlyFileImpl::read_property_ascii(const Type &t,
                                                 const size_t &stride,
                                                 void *dest, size_t &destOffset,
                                                 std::istream &is) {
  destOffset += stride;
  switch (t) {
    case Type::INT8:
      *((int8_t *)dest) = static_cast<int8_t>(ply_read_ascii<int32_t>(is));
      break;
    case Type::UINT8:
      *((uint8_t *)dest) = static_cast<uint8_t>(ply_read_ascii<uint32_t>(is));
      break;
    case Type::INT16:
      ply_cast_ascii<int16_t>(dest, is);
      break;
    case Type::UINT16:
```

(continues on next page)

(continued from previous page)

```
      ply_cast_ascii<uint16_t>(dest, is);
      break;
    case Type::INT32:
      ply_cast_ascii<int32_t>(dest, is);
      break;
    case Type::UINT32:
      ply_cast_ascii<uint32_t>(dest, is);
      break;
    case Type::FLOAT32:
      ply_cast_ascii<float>(dest, is);
      break;
    case Type::FLOAT64:
      ply_cast_ascii<double>(dest, is);
      break;
    case Type::INVALID:
      throw std::invalid_argument("invalid ply property");
  }
  return stride;
}

void PlyFile::PlyFileImpl::write_property_ascii(Type t, std::ostream &os,
                                                const uint8_t *src,
                                                size_t &srcOffset) {
  switch (t) {
    case Type::INT8:
      os << static_cast<int32_t>(*reinterpret_cast<const int8_t *>(src));
      break;
    case Type::UINT8:
      os << static_cast<uint32_t>(*reinterpret_cast<const uint8_t *>(src));
      break;
    case Type::INT16:
      os << *reinterpret_cast<const int16_t *>(src);
      break;
    case Type::UINT16:
      os << *reinterpret_cast<const uint16_t *>(src);
      break;
    case Type::INT32:
      os << *reinterpret_cast<const int32_t *>(src);
      break;
    case Type::UINT32:
      os << *reinterpret_cast<const uint32_t *>(src);
      break;
    case Type::FLOAT32:
      os << *reinterpret_cast<const float *>(src);
      break;
    case Type::FLOAT64:
      os << *reinterpret_cast<const double *>(src);
      break;
    case Type::INVALID:
      throw std::invalid_argument("invalid ply property");
  }
  os << " ";
```

(continues on next page)

(continued from previous page)

```
  srcOffset += PropertyTable[t].stride;
}

void PlyFile::PlyFileImpl::write_property_binary(
  std::ostream &os, const uint8_t *src, size_t &srcOffset,
  const size_t &stride) noexcept {
  os.write((char *)src, stride);
  srcOffset += stride;
}

void PlyFile::PlyFileImpl::read(std::istream &is) {
  std::vector<std::shared_ptr<PlyData>> buffers;
  for (auto &entry : userData) buffers.push_back(entry.second.data);

  // Discover if we can allocate up front without parsing the file twice
  uint32_t list_hints = 0;
  for (auto &b : buffers)
    for (auto &entry : userData) {
      list_hints += entry.second.list_size_hint;
      (void)b;
    }

  // No list hints? Then we need to calculate how much memory to allocate
  if (list_hints == 0) {
    parse_data(is, true);
  }

  // Count the number of properties (required for allocation)
  // e.g. if we have properties x y and z requested, we ensure
  // that their buffer points to the same PlyData
  std::unordered_map<PlyData *, int32_t> unique_data_count;
  for (auto &ptr : buffers) unique_data_count[ptr.get()] += 1;

  // Since group-requested properties share the same cursor,
  // we need to find unique cursors so we only allocate once
  std::sort(buffers.begin(), buffers.end());
  buffers.erase(std::unique(buffers.begin(), buffers.end()), buffers.end());

  // We sorted by ptrs on PlyData, need to remap back onto its cursor in the
  // userData table
  for (auto &b : buffers) {
    for (auto &entry : userData) {
      if (entry.second.data == b && b->buffer.get() == nullptr) {
        // If we didn't receive any list hints, it means we did two passes over
        // the file to compute the total length of all (potentially)
        // variable-length lists
        if (list_hints == 0) {
          b->buffer = Buffer(entry.second.cursor->totalSizeBytes);
        } else {
          // otherwise, we can allocate up front, skipping the first pass.
          const size_t list_size_multiplier =
            (entry.second.data->isList ? entry.second.list_size_hint : 1);
```

(continues on next page)

(continued from previous page)

```
          auto bytes_per_property = entry.second.data->count *
                                    PropertyTable[entry.second.data->t].stride *
                                    list_size_multiplier;
          bytes_per_property *= unique_data_count[b.get()];
          b->buffer = Buffer(bytes_per_property);
        }
      }
    }
  }

  // Populate the data
  parse_data(is, false);

  // In-place big-endian to little-endian swapping if required
  if (isBigEndian) {
    for (auto &b : buffers) {
      uint8_t *data_ptr = b->buffer.get();
      const size_t stride = PropertyTable[b->t].stride;
      const size_t buffer_size_bytes = b->buffer.size_bytes();

      switch (b->t) {
        case Type::INT16:
          endian_swap_buffer<int16_t, int16_t>(data_ptr, buffer_size_bytes,
                                               stride);
          break;
        case Type::UINT16:
          endian_swap_buffer<uint16_t, uint16_t>(data_ptr, buffer_size_bytes,
                                                 stride);
          break;
        case Type::INT32:
          endian_swap_buffer<int32_t, int32_t>(data_ptr, buffer_size_bytes,
                                               stride);
          break;
        case Type::UINT32:
          endian_swap_buffer<uint32_t, uint32_t>(data_ptr, buffer_size_bytes,
                                                 stride);
          break;
        case Type::FLOAT32:
          endian_swap_buffer<uint32_t, float>(data_ptr, buffer_size_bytes,
                                              stride);
          break;
        case Type::FLOAT64:
          endian_swap_buffer<uint64_t, double>(data_ptr, buffer_size_bytes,
                                               stride);
          break;
        default:
          break;
      }
    }
  }
}
```

(continues on next page)

(continued from previous page)

```
void PlyFile::PlyFileImpl::write(std::ostream &os, bool _isBinary) {
  for (auto &d : userData) {
    d.second.cursor->byteOffset = 0;
  }
  if (_isBinary) {
    isBinary = true;
    isBigEndian = false;
    write_binary_internal(os);
  } else {
    isBinary = false;
    isBigEndian = false;
    write_ascii_internal(os);
  }
}

void PlyFile::PlyFileImpl::write_binary_internal(std::ostream &os) noexcept {
  isBinary = true;

  write_header(os);

  uint8_t listSize[4] = {0, 0, 0, 0};
  size_t dummyCount = 0;

  auto element_property_lookup = make_property_lookup_table();

  size_t element_idx = 0;
  for (auto &e : elements) {
    for (size_t i = 0; i < e.size; ++i) {
      size_t property_index = 0;
      for (auto &p : e.properties) {
        auto &f = element_property_lookup[element_idx][property_index];
        auto *helper = f.helper;
        if (f.skip || helper == nullptr) continue;

        if (p.isList) {
          std::memcpy(listSize, &p.listCount, sizeof(uint32_t));
          write_property_binary(os, listSize, dummyCount, f.list_stride);
          write_property_binary(
            os, (helper->data->buffer.get_const() + helper->cursor->byteOffset),
            helper->cursor->byteOffset, f.prop_stride * p.listCount);
        } else {
          write_property_binary(
            os, (helper->data->buffer.get_const() + helper->cursor->byteOffset),
            helper->cursor->byteOffset, f.prop_stride);
        }
        property_index++;
      }
    }
    element_idx++;
  }
}
```

(continues on next page)

(continued from previous page)

```
void PlyFile::PlyFileImpl::write_ascii_internal(std::ostream &os) noexcept {
  write_header(os);

  auto element_property_lookup = make_property_lookup_table();

  size_t element_idx = 0;
  for (auto &e : elements) {
    for (size_t i = 0; i < e.size; ++i) {
      size_t property_index = 0;
      for (auto &p : e.properties) {
        auto &f = element_property_lookup[element_idx][property_index];
        auto *helper = f.helper;
        if (f.skip || helper == nullptr) continue;

        if (p.isList) {
          os << p.listCount << " ";
          for (size_t j = 0; j < p.listCount; ++j) {
            write_property_ascii(
              p.propertyType, os,
              (helper->data->buffer.get() + helper->cursor->byteOffset),
              helper->cursor->byteOffset);
          }
        } else {
          write_property_ascii(
            p.propertyType, os,
            (helper->data->buffer.get() + helper->cursor->byteOffset),
            helper->cursor->byteOffset);
        }
        property_index++;
      }
      os << "\n";
    }
    element_idx++;
  }
}

void PlyFile::PlyFileImpl::write_header(std::ostream &os) noexcept {
  const std::locale &fixLoc = std::locale("C");
  os.imbue(fixLoc);

  os << "ply\n";
  if (isBinary)
    os << ((isBigEndian) ? "format binary_big_endian 1.0"
                         : "format binary_little_endian 1.0")
       << "\n";
  else
    os << "format ascii 1.0\n";

  for (const auto &comment : comments) os << "comment " << comment << "\n";

  auto property_lookup = make_property_lookup_table();
```

(continues on next page)

(continued from previous page)

```
  size_t element_idx = 0;
  for (auto &e : elements) {
    os << "element " << e.name << " " << e.size << "\n";
    size_t property_idx = 0;
    for (const auto &p : e.properties) {
      PropertyLookup &lookup = property_lookup[element_idx][property_idx];

      if (!lookup.skip) {
        if (p.isList) {
          os << "property list " << PropertyTable[p.listType].str << " "
             << PropertyTable[p.propertyType].str << " " << p.name << "\n";
        } else {
          os << "property " << PropertyTable[p.propertyType].str << " "
             << p.name << "\n";
        }
      }
      property_idx++;
    }
    element_idx++;
  }
  os << "end_header\n";
}

std::shared_ptr<PlyData> PlyFile::PlyFileImpl::request_properties_from_element(
  const std::string &elementKey, const std::vector<std::string> propertyKeys,
  const uint32_t list_size_hint) {
  if (elements.empty())
    throw std::runtime_error("header had no elements defined. malformed file?");
  if (elementKey.empty())
    throw std::invalid_argument("`elementKey` argument is empty");
  if (propertyKeys.empty())
    throw std::invalid_argument("`propertyKeys` argument is empty");

  std::shared_ptr<PlyData> out_data = std::make_shared<PlyData>();

  const int64_t elementIndex = find_element(elementKey, elements);

  std::vector<std::string> keys_not_found;

  // Sanity check if the user requested element is in the pre-parsed header
  if (elementIndex >= 0) {
    // We found the element
    const PlyElement &element = elements[elementIndex];

    // Each key in `propertyKey` gets an entry into the userData map (keyed by a
    // hash of element name and property name), but groups of properties
    // (requested from the public api through this function) all share the same
    // `ParsingHelper`. When it comes time to .read(), we check the number of
    // unique PlyData shared pointers and allocate a single buffer that will be
    // used by each property key group. That way, properties like, {"x", "y",
    // "z"} will all be put into the same buffer.
```

(continues on next page)

(continued from previous page)

```
    ParsingHelper helper;
    helper.data = out_data;
    helper.data->count = element.size;  // how many items are in the element?
    helper.data->isList = false;
    helper.data->t = Type::INVALID;
    helper.cursor = std::make_shared<PlyDataCursor>();
    helper.list_size_hint = list_size_hint;

    // Find each of the keys
    for (const auto &key : propertyKeys) {
      const int64_t propertyIndex = find_property(key, element.properties);
      if (propertyIndex < 0) keys_not_found.push_back(key);
    }

    if (keys_not_found.size()) {
      std::stringstream ss;
      for (auto &str : keys_not_found) ss << str << ", ";
      throw std::invalid_argument(
        "the following property keys were not found in the header: " +
        ss.str());
    }

    for (const auto &key : propertyKeys) {
      const int64_t propertyIndex = find_property(key, element.properties);
      const PlyProperty &property = element.properties[propertyIndex];
      helper.data->t = property.propertyType;
      helper.data->isList = property.isList;
      auto result = userData.insert(std::pair<uint32_t, ParsingHelper>(
        hash_fnv1a(element.name + property.name), helper));
      if (result.second == false) {
        throw std::invalid_argument(
          "element-property key has already been requested: " + element.name +
          " " + property.name);
      }
    }

    // Sanity check that all properties share the same type
    std::vector<Type> propertyTypes;
    for (const auto &key : propertyKeys) {
      const int64_t propertyIndex = find_property(key, element.properties);
      const PlyProperty &property = element.properties[propertyIndex];
      propertyTypes.push_back(property.propertyType);
    }

    if (std::adjacent_find(propertyTypes.begin(), propertyTypes.end(),
                           std::not_equal_to<Type>()) != propertyTypes.end()) {
      throw std::invalid_argument(
        "all requested properties must share the same type.");
    }
  } else
    throw std::invalid_argument(
      "the element key was not found in the header: " + elementKey);
```

(continues on next page)

(continued from previous page)

```
  return out_data;
}

void PlyFile::PlyFileImpl::add_properties_to_element(
  const std::string &elementKey, const std::vector<std::string> propertyKeys,
  const Type type, const size_t count, const uint8_t *data, const Type listType,
  const size_t listCount) {
  ParsingHelper helper;
  helper.data = std::make_shared<PlyData>();
  helper.data->count = count;
  helper.data->t = type;
  helper.data->buffer =
    Buffer(data);  // we should also set size for safety reasons
  helper.cursor = std::make_shared<PlyDataCursor>();

  auto create_property_on_element = [&](PlyElement &e) {
    for (auto key : propertyKeys) {
      PlyProperty newProp = (listType == Type::INVALID)
                              ? PlyProperty(type, key)
                              : PlyProperty(listType, type, key, listCount);
      userData.insert(std::pair<uint32_t, ParsingHelper>(
        hash_fnv1a(elementKey + key), helper));
      e.properties.push_back(newProp);
    }
  };

  const int64_t idx = find_element(elementKey, elements);
  if (idx >= 0) {
    PlyElement &e = elements[idx];
    create_property_on_element(e);
  } else {
    PlyElement newElement = (listType == Type::INVALID)
                              ? PlyElement(elementKey, count)
                              : PlyElement(elementKey, count);
    create_property_on_element(newElement);
    elements.push_back(newElement);
  }
}

void PlyFile::PlyFileImpl::parse_data(std::istream &is, bool firstPass) {
  std::function<void(PropertyLookup & f, const PlyProperty &p, uint8_t *dest,
                     size_t &destOffset, std::istream &is)>
    read;
  std::function<size_t(PropertyLookup & f, const PlyProperty &p,
                       std::istream &is)>
    skip;

  const auto start = is.tellg();

  uint32_t listSize = 0;
  size_t dummyCount = 0;
```

(continues on next page)

(continued from previous page)

```
  std::string skip_ascii_buffer;

  // Special case mirroring read_property_binary but for list types; this
  // has an additional big endian check to flip the data in place immediately
  // after reading. We do this as a performance optimization; endian flipping is
  // done on regular properties as a post-process after reading (also for
  // optimization) but we need the correct little-endian list count as we read
  // the file.
  auto read_list_binary = [this](const Type &t, void *dst, size_t &destOffset,
                                 const size_t &stride,
                                 std::istream &_is) noexcept {
    destOffset += stride;
    _is.read((char *)dst, stride);

    if (isBigEndian) {
      switch (t) {
        case Type::INT16:
          *(int16_t *)dst = endian_swap<int16_t, int16_t>(*(int16_t *)dst);
          break;
        case Type::UINT16:
          *(uint16_t *)dst = endian_swap<uint16_t, uint16_t>(*(uint16_t *)dst);
          break;
        case Type::INT32:
          *(int32_t *)dst = endian_swap<int32_t, int32_t>(*(int32_t *)dst);
          break;
        case Type::UINT32:
          *(uint32_t *)dst = endian_swap<uint32_t, uint32_t>(*(uint32_t *)dst);
          break;
        default:
          break;
      }
    }

    return stride;
  };

  if (isBinary) {
    read = [ this, &listSize, &dummyCount, &read_list_binary ](
      PropertyLookup & f, const PlyProperty &p, uint8_t *dest,
      size_t &destOffset, std::istream &_is) noexcept {
      if (!p.isList) {
        return read_property_binary(f.prop_stride, dest + destOffset,
                                    destOffset, _is);
      }
      read_list_binary(p.listType, &listSize, dummyCount, f.list_stride,
                       _is);  // the list size
      return read_property_binary(f.prop_stride * listSize, dest + destOffset,
                                  destOffset, _is);  // properties in list
    };
    skip = [ this, &listSize, &dummyCount, &read_list_binary ](
      PropertyLookup & f, const PlyProperty &p, std::istream &_is) noexcept {
      if (!p.isList) {
```

(continues on next page)

(continued from previous page)

```
      _is.read((char *)scratch, f.prop_stride);
      return f.prop_stride;
    }
    read_list_binary(p.listType, &listSize, dummyCount, f.list_stride,
                     _is);  // the list size (does not count for memory alloc)
    auto bytes_to_skip = f.prop_stride * listSize;
    _is.ignore(bytes_to_skip);
    return bytes_to_skip;
  };
} else {
  read = [ this, &listSize, &dummyCount ](
    PropertyLookup & f, const PlyProperty &p, uint8_t *dest,
    size_t &destOffset, std::istream &_is) noexcept {
    if (!p.isList) {
      read_property_ascii(p.propertyType, f.prop_stride, dest + destOffset,
                          destOffset, _is);
    } else {
      read_property_ascii(p.listType, f.list_stride, &listSize, dummyCount,
                          _is);  // the list size
      for (size_t i = 0; i < listSize; ++i) {
        read_property_ascii(p.propertyType, f.prop_stride, dest + destOffset,
                            destOffset, _is);
      }
    }
  };
  skip = [ this, &listSize, &dummyCount, &skip_ascii_buffer ](
    PropertyLookup & f, const PlyProperty &p, std::istream &_is) noexcept {
    skip_ascii_buffer.clear();
    if (p.isList) {
      read_property_ascii(
        p.listType, f.list_stride, &listSize, dummyCount,
        _is);  // the list size (does not count for memory alloc)
      for (size_t i = 0; i < listSize; ++i)
        _is >> skip_ascii_buffer;  // properties in list
      return listSize * f.prop_stride;
    }
    _is >> skip_ascii_buffer;
    return f.prop_stride;
  };
}

std::vector<std::vector<PropertyLookup>> element_property_lookup =
  make_property_lookup_table();
size_t element_idx = 0;
size_t property_idx = 0;
ParsingHelper *helper{nullptr};

// This is the inner import loop
for (auto &element : elements) {
  for (size_t count = 0; count < element.size; ++count) {
    property_idx = 0;
    for (auto &property : element.properties) {
```

(continues on next page)

(continued from previous page)

```
        PropertyLookup &lookup =
          element_property_lookup[element_idx][property_idx];

        if (!lookup.skip) {
          helper = lookup.helper;
          if (firstPass) {
            helper->cursor->totalSizeBytes += skip(lookup, property, is);

            // These lines will be changed when tinyply supports
            // variable length lists. We add it here so our header data
            // structure contains enough info to write it back out again (e.g.
            // transcoding).
            if (property.listCount == 0) property.listCount = listSize;
            if (property.listCount != listSize)
              throw std::runtime_error(
                "variable length lists are not supported yet.");
          } else {
            read(lookup, property, helper->data->buffer.get(),
                 helper->cursor->byteOffset, is);
          }
        } else {
          skip(lookup, property, is);
        }
        property_idx++;
      }
    }
    element_idx++;
  }

  // Reset istream position to the start of the data
  if (firstPass) is.seekg(start, is.beg);
}

// Wrap the public interface:

PlyFile::PlyFile() { impl.reset(new PlyFileImpl()); }
PlyFile::~PlyFile() {}
bool PlyFile::parse_header(std::istream &is) { return impl->parse_header(is); }
void PlyFile::read(std::istream &is) { return impl->read(is); }
void PlyFile::write(std::ostream &os, bool isBinary) {
  return impl->write(os, isBinary);
}
std::vector<PlyElement> PlyFile::get_elements() const { return impl->elements; }
std::vector<std::string> &PlyFile::get_comments() { return impl->comments; }
std::vector<std::string> PlyFile::get_info() const { return impl->objInfo; }
bool PlyFile::is_binary_file() const { return impl->isBinary; }
std::shared_ptr<PlyData> PlyFile::request_properties_from_element(
  const std::string &elementKey, const std::vector<std::string> propertyKeys,
  const uint32_t list_size_hint) {
  return impl->request_properties_from_element(elementKey, propertyKeys,
                                               list_size_hint);
}
```

(continues on next page)

(continued from previous page)

```
void PlyFile::add_properties_to_element(
  const std::string &elementKey, const std::vector<std::string> propertyKeys,
  const Type type, const size_t count, const uint8_t *data, const Type listType,
  const size_t listCount) {
  return impl->add_properties_to_element(elementKey, propertyKeys, type, count,
                                         data, listType, listCount);
}

#endif  // end TINYPLY_IMPLEMENTATION
```

Header

```
#pragma once

// standard libraries
#include <assert.h>
#include <Eigen/Dense>
#include <chrono>
#include <cmath>
#include <cstring>
#include <experimental/filesystem>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <thread>
#include <vector>

// Open3D
#include <Open3D/Geometry/KDTreeFlann.h>
#include <Open3D/Geometry/PointCloud.h>
#include <Open3D/IO/ClassIO/PointCloudIO.h>

// OMPL
#include <ompl/base/SpaceInformation.h>
#include <ompl/base/spaces/SE3StateSpace.h>
#include <ompl/config.h>
#include <ompl/geometric/SimpleSetup.h>
#include <ompl/geometric/planners/rrt/RRTConnect.h>

// TinyPly
#define TINYPLY_IMPLEMENTATION
#include "tinyply.h"

// flightlib
#include "flightlib/bridges/unity_bridge.hpp"
#include "flightlib/bridges/unity_message_types.hpp"
#include "flightlib/common/quad_state.hpp"
#include "flightlib/common/types.hpp"
#include "flightlib/objects/quadrotor.hpp"
```

(continues on next page)

(continued from previous page)

```
#include "flightlib/sensors/rgb_camera.hpp"

// ros
#include <ros/ros.h>

namespace ob = ompl::base;
namespace og = ompl::geometric;

using namespace flightlib;

namespace motion_planning {

class manual_timer {
  std::chrono::high_resolution_clock::time_point t0;
  double timestamp{0.0};

 public:
  void start() { t0 = std::chrono::high_resolution_clock::now(); }
  void stop() {
    timestamp = std::chrono::duration<double>(
                  std::chrono::high_resolution_clock::now() - t0)
                  .count() *
                1000.0;
  }
  const double &get() { return timestamp; }
};

struct float2 {
  float x, y;
};
struct float3 {
  float x, y, z;
};
struct double3 {
  double x, y, z;
};
struct uint3 {
  uint32_t x, y, z;
};
struct uint4 {
  uint32_t x, y, z, w;
};
std::vector<float3> verts;
float range = 1;
bool solution_found = false;
bool trajectory_found = false;

std::vector<float3> readPointCloud();
float3 min_bounds;
float3 max_bounds;

Eigen::Vector3d stateToEigen(const ompl::base::State *state);
```

(continues on next page)

(continued from previous page)

```
std::vector<ompl::base::State *> path_;
std::vector<Eigen::Vector3d> vecs_;

void getBounds();

void plan();

bool isStateValid(const ob::State *state);

bool isInRange(float x, float y, float z);

open3d::geometry::KDTreeFlann kd_tree_;
Eigen::MatrixXd points_;
bool searchRadius(const Eigen::Vector3d &query_point, const double radius);

void executePath();

// void setupQuad();
bool setUnity(const bool render);
bool connectUnity(void);

// unity quadrotor
std::shared_ptr<Quadrotor> quad_ptr_;
std::shared_ptr<RGBCamera> rgb_camera_;
QuadState quad_state_;

// Flightmare(Unity3D)
std::shared_ptr<UnityBridge> unity_bridge_ptr_;
SceneID scene_id_{UnityScene::NATUREFOREST};
bool unity_ready_{false};
bool unity_render_{true};
RenderMessage_t unity_output_;
uint16_t receive_id_{0};


}  // namespace motion_planning
```

Main

```
#include "flightros/motion_planning/motion_planning.hpp"

#define CONTROL_UPDATE_RATE 50.0

#define TINYPLY_IMPLEMENTATION

namespace ob = ompl::base;
namespace og = ompl::geometric;
using namespace tinyply;
```

(continues on next page)

(continued from previous page)

```
std::vector<motion_planning::float3> motion_planning::readPointCloud() {
  std::unique_ptr<std::istream> file_stream;
  std::vector<uint8_t> byte_buffer;
  std::string filepath =
    std::experimental::filesystem::path(__FILE__).parent_path().string() +
    "/data/point_cloud.ply";
  try {
    file_stream.reset(new std::ifstream(filepath, std::ios::binary));

    if (!file_stream || file_stream->fail())
      throw std::runtime_error("file_stream failed to open " + filepath);

    file_stream->seekg(0, std::ios::end);
    const float size_mb = file_stream->tellg() * float(1e-6);
    file_stream->seekg(0, std::ios::beg);

    PlyFile file;
    file.parse_header(*file_stream);

    std::cout << "\t[ply_header] Type: "
              << (file.is_binary_file() ? "binary" : "ascii") << std::endl;
    for (const auto &c : file.get_comments())
      std::cout << "\t[ply_header] Comment: " << c << std::endl;
    for (const auto &c : file.get_info())
      std::cout << "\t[ply_header] Info: " << c << std::endl;

    for (const auto &e : file.get_elements()) {
      std::cout << "\t[ply_header] element: " << e.name << " (" << e.size << ")"
                << std::endl;
      for (const auto &p : e.properties) {
        std::cout << "\t[ply_header] \tproperty: " << p.name
                  << " (type=" << tinyply::PropertyTable[p.propertyType].str
                  << ")";
        if (p.isList)
          std::cout << " (list_type=" << tinyply::PropertyTable[p.listType].str
                    << ")";
        std::cout << std::endl;
      }
    }

    // Because most people have their own mesh types, tinyply treats parsed data
    // as structured/typed byte buffers. See examples below on how to marry your
    // own application-specific data structures with this one.
    std::shared_ptr<PlyData> vertices, normals, colors, texcoords, faces,
      tripstrip;

    // The header information can be used to programmatically extract properties
    // on elements known to exist in the header prior to reading the data. For
    // brevity of this sample, properties like vertex position are hard-coded:
    try {
      vertices =
        file.request_properties_from_element("vertex", {"x", "y", "z"});
```

(continues on next page)

(continued from previous page)

```
    } catch (const std::exception &e) {
      std::cerr << "tinyply exception: " << e.what() << std::endl;
    }

    manual_timer read_timer;

    read_timer.start();
    file.read(*file_stream);
    read_timer.stop();

    const float parsing_time = read_timer.get() / 1000.f;
    std::cout << "\tparsing " << size_mb << "mb in " << parsing_time
              << " seconds [" << (size_mb / parsing_time) << " MBps]"
              << std::endl;

    if (vertices)
      std::cout << "\tRead " << vertices->count << " total vertices "
                << std::endl;


    const size_t numVerticesBytes = vertices->buffer.size_bytes();
    std::vector<float3> verts(vertices->count);
    std::memcpy(verts.data(), vertices->buffer.get(), numVerticesBytes);

    int idx = 0;
    for (const auto &point_tinyply : verts) {
      if (idx == 0) {
        points_ = Eigen::Vector3d(static_cast<double>(point_tinyply.x),
                                  static_cast<double>(point_tinyply.y),
                                  static_cast<double>(point_tinyply.z));
      } else {
        points_.conservativeResize(points_.rows(), points_.cols() + 1);
        points_.col(points_.cols() - 1) =
          Eigen::Vector3d(static_cast<double>(point_tinyply.x),
                          static_cast<double>(point_tinyply.y),
                          static_cast<double>(point_tinyply.z));
      }
      idx += 1;
    }
    kd_tree_.SetMatrixData(points_);

    return verts;
  } catch (const std::exception &e) {
    std::cerr << "Caught tinyply exception: " << e.what() << std::endl;
  }
  return verts;
}

void motion_planning::getBounds() {
  min_bounds = verts[0];
  max_bounds = verts[0];
```

(continues on next page)

(continued from previous page)

```
  for (const auto &ver : verts) {
    if (ver.x < min_bounds.x) {
      min_bounds.x = ver.x;
    }

    if (ver.x > max_bounds.x) {
      max_bounds.x = ver.x;
    }

    if (ver.y < min_bounds.y) {
      min_bounds.y = ver.y;
    }

    if (ver.y > max_bounds.y) {
      max_bounds.y = ver.y;
    }

    if (ver.z > max_bounds.z) {
      max_bounds.z = ver.z;
    }
  }
}


void motion_planning::plan() {
  // construct the state space we are planning in
  auto space(std::make_shared<ob::SE3StateSpace>());

  // set the bounds for the R^3 part of SE(3)
  ob::RealVectorBounds bounds(3);

  bounds.setLow(0, min_bounds.x);
  bounds.setLow(1, min_bounds.y);
  bounds.setLow(2, min_bounds.z);
  bounds.setHigh(0, max_bounds.x);
  bounds.setHigh(1, max_bounds.y);
  bounds.setHigh(2, max_bounds.z);

  space->setBounds(bounds);

  // define a simple setup class
  og::SimpleSetup ss(space);

  // set state validity checking for this space
  ss.setStateValidityChecker(
    [](const ob::State *state) { return isStateValid(state); });

  // create a random start state
  ob::ScopedState<> start(space);
  start.random();
```

(continues on next page)

(continued from previous page)

```
  // create a random goal state
  ob::ScopedState<> goal(space);
  goal.random();

  // set the start and goal states
  ss.setStartAndGoalStates(start, goal);

  // this call is optional, but we put it in to get more output information
  ss.setup();
  ss.print();

  // attempt to solve the problem within one second of planning time
  ob::PlannerStatus solved = ss.solve(1.0);

  if (solved) {
    std::cout << "Found solution:" << std::endl;

    ob::PlannerData pd(ss.getSpaceInformation());
    ss.getPlannerData(pd);
    /** backup cout buffer and redirect to path.txt **/
    std::ofstream out0(
      std::experimental::filesystem::path(__FILE__).parent_path().string() +
      "/data//vertices.txt");
    auto *coutbuf0 = std::cout.rdbuf();
    std::cout.rdbuf(out0.rdbuf());


    unsigned int num_vertices = pd.numVertices();
    for (unsigned int i = 0; i < num_vertices; i++) {
      Eigen::Vector3d p = stateToEigen(pd.getVertex(i).getState());
      std::cout << p.x() << " " << p.y() << " " << p.z() << " " << std::endl;
    }

    /** reset cout buffer **/
    std::cout.rdbuf(coutbuf0);

    /** backup cout buffer and redirect to path.txt **/
    std::ofstream out00(
      std::experimental::filesystem::path(__FILE__).parent_path().string() +
      "/data/edges.txt");
    auto *coutbuf00 = std::cout.rdbuf();
    std::cout.rdbuf(out00.rdbuf());

    for (unsigned int i = 0; i < num_vertices; i++) {
      std::vector<unsigned int> e;
      pd.getEdges(i, e);
      for (const auto &j : e) {
        std::cout << i << " " << j << " " << std::endl;
      }
    }

    /** reset cout buffer **/
```

(continues on next page)

(continued from previous page)

```
    std::cout.rdbuf(coutbuf00);

    // save solution in solution_path.txt
    /** backup cout buffer and redirect to path.txt **/
    ss.simplifySolution();
    std::ofstream out(
      std::experimental::filesystem::path(__FILE__).parent_path().string() +
      "/data/solution_path.txt");
    auto *coutbuf = std::cout.rdbuf();
    std::cout.rdbuf(out.rdbuf());
    ss.getSolutionPath().printAsMatrix(std::cout);
    /** reset cout buffer **/
    std::cout.rdbuf(coutbuf);
    ss.getSolutionPath().print(std::cout);
    path_.clear();
    path_ = ss.getSolutionPath().getStates();
    vecs_.clear();
    for (auto const &pos : path_) {
      vecs_.push_back(stateToEigen(pos));
    }

    if (path_.size() > 1) {
      solution_found = true;
    }
  } else
    std::cout << "No solution found" << std::endl;
}

Eigen::Vector3d motion_planning::stateToEigen(const ompl::base::State *state) {
  // cast the abstract state type to the type we expect
  const auto *se3state = state->as<ob::SE3StateSpace::StateType>();

  // extract the first component of the state and cast it to what we expect
  const auto *pos = se3state->as<ob::RealVectorStateSpace::StateType>(0);

  // extract the second component of the state and cast it to what we expect
  //  const auto *rot = se3state->as<ob::SO3StateSpace::StateType>(1);

  // check validity of state defined by pos & rot
  float x = pos->values[0];
  float y = pos->values[1];
  float z = pos->values[2];

  // return a value that is always true but uses the two variables we define, so
  // we avoid compiler warnings
  // return isInRange(x, y, z);
  Eigen::Vector3d query_pos{x, y, z};
  return query_pos;
}

bool motion_planning::isStateValid(const ob::State *state) {
  // cast the abstract state type to the type we expect
```

(continues on next page)

(continued from previous page)

```
  const auto *se3state = state->as<ob::SE3StateSpace::StateType>();

  // extract the first component of the state and cast it to what we expect
  const auto *pos = se3state->as<ob::RealVectorStateSpace::StateType>(0);

  // extract the second component of the state and cast it to what we expect
  //  const auto *rot = se3state->as<ob::SO3StateSpace::StateType>(1);

  // check validity of state defined by pos & rot
  float x = pos->values[0];
  float y = pos->values[1];
  float z = pos->values[2];
  // return a value that is always true but uses the two variables we define, so
  // we avoid compiler warnings
  // return isInRange(x, y, z);
  Eigen::Vector3d query_pos{x, y, z};
  return searchRadius(query_pos, range);
}

bool motion_planning::isInRange(float x, float y, float z) {
  bool validState = true;
  bool outOfBound = false;

  for (const auto &ver : verts) {
    // check if box around quad is occupied
    if (abs(ver.z - z) <= range) {
      if (abs(ver.x - x) <= range) {
        if (abs(ver.y - y) <= range) {
          validState = false;
        }
      }
    } else {
      if (ver.z - z >= 0) {
        outOfBound = true;
      }
    }


    if (outOfBound || !validState) {
      break;
    }
  }
  return validState;
}


bool motion_planning::searchRadius(const Eigen::Vector3d &query_point,
                                   const double radius) {
  std::vector<int> indices;
  std::vector<double> distances_squared;
  kd_tree_.SearchRadius(query_point, radius, indices, distances_squared);
```

(continues on next page)

(continued from previous page)

```
  if (indices.size() == 0) {
    return true;
  }

  for (const auto &close_point_idx : indices) {
    // get point, check if within drone body
    Eigen::Vector3d close_point = points_.col(close_point_idx);
    // project point on each body axis and check distance
    Eigen::Vector3d close_point_body = (close_point - query_point);
    if (std::abs(close_point_body.x()) <= range &&
        std::abs(close_point_body.y()) <= range &&
        std::abs(close_point_body.z()) <= range) {
      // point is in collision
      return false;
    }
  }

  return true;
}

void motion_planning::executePath() {
  // initialization
  assert(!vecs_.empty());
  assert(vecs_.at(0).x() != NULL);

  // compute trajectory
  std::size_t num_waypoints = vecs_.size();
  int scaling = 10;

  std::vector<Eigen::Vector3d> way_points;

  for (int i = 0; i < int(num_waypoints - 1); i++) {
    Eigen::Vector3d diff_vec = vecs_.at(i + 1) - vecs_.at(i);
    double norm = diff_vec.norm();
    for (int j = 0; j < int(norm * scaling); j++) {
      way_points.push_back(vecs_.at(i) + j * (diff_vec / (norm * scaling)));
    }
  }

  // Flightmare
  Vector<3> B_r_BC(0.0, 0.0, 0.3);
  Matrix<3, 3> R_BC = Quaternion(1.0, 0.0, 0.0, 0.0).toRotationMatrix();
  std::cout << R_BC << std::endl;
  rgb_camera_->setFOV(90);
  rgb_camera_->setWidth(720);
  rgb_camera_->setHeight(480);
  rgb_camera_->setRelPose(B_r_BC, R_BC);
  quad_ptr_->addRGBCamera(rgb_camera_);

  quad_state_.setZero();
  quad_state_.x[QS::POSX] = (Scalar)vecs_.at(0).x();
  quad_state_.x[QS::POSY] = (Scalar)vecs_.at(0).y();
```

(continues on next page)

(continued from previous page)

```
  quad_state_.x[QS::POSZ] = (Scalar)vecs_.at(0).z();

  quad_ptr_->reset(quad_state_);

  // connect unity
  setUnity(unity_render_);
  connectUnity();
  assert(unity_ready_);
  assert(unity_render_);

  int counter = 0;

  while (unity_render_ && unity_ready_) {
    quad_state_.x[QS::POSX] = (Scalar)way_points.at(counter).x();
    quad_state_.x[QS::POSY] = (Scalar)way_points.at(counter).y();
    quad_state_.x[QS::POSZ] = (Scalar)way_points.at(counter).z();
    quad_state_.x[QS::ATTW] = (Scalar)0;
    quad_state_.x[QS::ATTX] = (Scalar)0;
    quad_state_.x[QS::ATTY] = (Scalar)0;
    quad_state_.x[QS::ATTZ] = (Scalar)0;

    quad_ptr_->setState(quad_state_);

    unity_bridge_ptr_->getRender(0);
    unity_bridge_ptr_->handleOutput();

    counter += 1;
    counter = counter % way_points.size();
  }
}


bool motion_planning::setUnity(const bool render) {
  unity_render_ = render;
  if (unity_render_ && unity_bridge_ptr_ == nullptr) {
    // create unity bridge
    unity_bridge_ptr_ = UnityBridge::getInstance();
    unity_bridge_ptr_->addQuadrotor(quad_ptr_);
    std::cout << "Unity Bridge is created." << std::endl;
  }
  return true;
}

bool motion_planning::connectUnity() {
  if (!unity_render_ || unity_bridge_ptr_ == nullptr) return false;
  unity_ready_ = unity_bridge_ptr_->connectUnity(scene_id_);
  return unity_ready_;
}


int main(int argc, char *argv[]) {
  // initialize ROS
```

(continues on next page)

(continued from previous page)

```
  ros::init(argc, argv, "flightmare_example");
  ros::NodeHandle nh("");
  ros::NodeHandle pnh("~");

  // quad initialization
  motion_planning::quad_ptr_ = std::make_unique<Quadrotor>();
  // add camera
  motion_planning::rgb_camera_ = std::make_unique<RGBCamera>();

  std::cout << "Read PointCloud" << std::endl;
  motion_planning::verts = motion_planning::readPointCloud();

  std::cout << "Get Bounds" << std::endl;
  motion_planning::getBounds();

  std::cout << "Plan & stuff" << std::endl;
  while (!motion_planning::solution_found) {
    motion_planning::plan();
  }

  std::cout << "Execute" << std::endl;
  motion_planning::executePath();


  return 0;
}
```

10.16 Pilot Tutorial

Simulate the quadrotor with your dynamic model while using Flightmare to generate sensor data.

10.16.1 Use Gazebo dynamics

Listen to ROS topic and set state

```
// initialize subscriber call backs
sub_state_est_ = nh_.subscribe("flight_pilot/state_estimate", 1,
                               &FlightPilot::poseCallback, this);

void FlightPilot::poseCallback(const nav_msgs::Odometry::ConstPtr &msg) {
  quad_state_.x[QS::POSX] = (Scalar)msg->pose.pose.position.x;
  quad_state_.x[QS::POSY] = (Scalar)msg->pose.pose.position.y;
  quad_state_.x[QS::POSZ] = (Scalar)msg->pose.pose.position.z;
  quad_state_.x[QS::ATTW] = (Scalar)msg->pose.pose.orientation.w;
  quad_state_.x[QS::ATTX] = (Scalar)msg->pose.pose.orientation.x;
  quad_state_.x[QS::ATTY] = (Scalar)msg->pose.pose.orientation.y;
  quad_state_.x[QS::ATTZ] = (Scalar)msg->pose.pose.orientation.z;
  //
```

(continues on next page)

(continued from previous page)

```
  quad_ptr_->setState(quad_state_);

  if (unity_render_ && unity_ready_) {
    unity_bridge_ptr_->getRender(0);
    unity_bridge_ptr_->handleOutput();
  }
}
```

10.16.2 Run example

```
roslaunch flightros rotors_gazebo.launch
```

10.16.3 Here the full code example

Header

```
#pragma once

#include <memory>

// ros
#include <nav_msgs/Odometry.h>
#include <ros/ros.h>

// rpg quadrotor
#include <autopilot/autopilot_helper.h>
#include <autopilot/autopilot_states.h>
#include <quadrotor_common/parameter_helper.h>
#include <quadrotor_msgs/AutopilotFeedback.h>

// flightlib
#include "flightlib/bridges/unity_bridge.hpp"
#include "flightlib/common/quad_state.hpp"
#include "flightlib/common/types.hpp"
#include "flightlib/objects/quadrotor.hpp"
#include "flightlib/sensors/rgb_camera.hpp"

using namespace flightlib;

namespace flightros {

class FlightPilot {
 public:
  EIGEN_MAKE_ALIGNED_OPERATOR_NEW
  FlightPilot(const ros::NodeHandle& nh, const ros::NodeHandle& pnh);
  ~FlightPilot();

  // callbacks
```

(continues on next page)

(continued from previous page)

```
  void mainLoopCallback(const ros::TimerEvent& event);
  void poseCallback(const nav_msgs::Odometry::ConstPtr& msg);

  bool setUnity(const bool render);
  bool connectUnity(void);
  bool loadParams(void);

 private:
  // ros nodes
  ros::NodeHandle nh_;
  ros::NodeHandle pnh_;

  // publisher

  // subscriber
  ros::Subscriber sub_state_est_;

  // main loop timer
  ros::Timer timer_main_loop_;

  // unity quadrotor
  std::shared_ptr<Quadrotor> quad_ptr_;
  std::shared_ptr<RGBCamera> rgb_camera_;
  QuadState quad_state_;

  // Flightmare(Unity3D)
  std::shared_ptr<UnityBridge> unity_bridge_ptr_;
  SceneID scene_id_{UnityScene::WAREHOUSE};
  bool unity_ready_{false};
  bool unity_render_{false};
  RenderMessage_t unity_output_;
  uint16_t receive_id_{0};

  // auxiliary variables
  Scalar main_loop_freq_{50.0};
};
}  // namespace flightros
```

Main

```
#include "flightros/pilot/flight_pilot.hpp"

namespace flightros {

FlightPilot::FlightPilot(const ros::NodeHandle &nh, const ros::NodeHandle &pnh)
  : nh_(nh),
    pnh_(pnh),
    scene_id_(UnityScene::WAREHOUSE),
    unity_ready_(false),
    unity_render_(false),
```

(continues on next page)

(continued from previous page)

```
    receive_id_(0),
    main_loop_freq_(50.0) {
  // load parameters
  if (!loadParams()) {
    ROS_WARN("[%s] Could not load all parameters.",
             pnh_.getNamespace().c_str());
  } else {
    ROS_INFO("[%s] Loaded all parameters.", pnh_.getNamespace().c_str());
  }

  // quad initialization
  quad_ptr_ = std::make_shared<Quadrotor>();

  // add mono camera
  rgb_camera_ = std::make_shared<RGBCamera>();
  Vector<3> B_r_BC(0.0, 0.0, 0.3);
  Matrix<3, 3> R_BC = Quaternion(1.0, 0.0, 0.0, 0.0).toRotationMatrix();
  std::cout << R_BC << std::endl;
  rgb_camera_->setFOV(90);
  rgb_camera_->setWidth(720);
  rgb_camera_->setHeight(480);
  rgb_camera_->setRelPose(B_r_BC, R_BC);
  quad_ptr_->addRGBCamera(rgb_camera_);

  // initialization
  quad_state_.setZero();
  quad_ptr_->reset(quad_state_);


  // initialize subscriber call backs
  sub_state_est_ = nh_.subscribe("flight_pilot/state_estimate", 1,
                                 &FlightPilot::poseCallback, this);

  timer_main_loop_ = nh_.createTimer(ros::Rate(main_loop_freq_),
                                     &FlightPilot::mainLoopCallback, this);


  // wait until the gazebo and unity are loaded
  ros::Duration(5.0).sleep();

  // connect unity
  setUnity(unity_render_);
  connectUnity();
}

FlightPilot::~FlightPilot() {}

void FlightPilot::poseCallback(const nav_msgs::Odometry::ConstPtr &msg) {
  quad_state_.x[QS::POSX] = (Scalar)msg->pose.pose.position.x;
  quad_state_.x[QS::POSY] = (Scalar)msg->pose.pose.position.y;
  quad_state_.x[QS::POSZ] = (Scalar)msg->pose.pose.position.z;
  quad_state_.x[QS::ATTW] = (Scalar)msg->pose.pose.orientation.w;
```

(continues on next page)

(continued from previous page)

```
  quad_state_.x[QS::ATTX] = (Scalar)msg->pose.pose.orientation.x;
  quad_state_.x[QS::ATTY] = (Scalar)msg->pose.pose.orientation.y;
  quad_state_.x[QS::ATTZ] = (Scalar)msg->pose.pose.orientation.z;
  //
  quad_ptr_->setState(quad_state_);

  if (unity_render_ && unity_ready_) {
    unity_bridge_ptr_->getRender(0);
    unity_bridge_ptr_->handleOutput();
  }
}

void FlightPilot::mainLoopCallback(const ros::TimerEvent &event) {
  // empty
}

bool FlightPilot::setUnity(const bool render) {
  unity_render_ = render;
  if (unity_render_ && unity_bridge_ptr_ == nullptr) {
    // create unity bridge
    unity_bridge_ptr_ = UnityBridge::getInstance();
    unity_bridge_ptr_->addQuadrotor(quad_ptr_);
    ROS_INFO("[%s] Unity Bridge is created.", pnh_.getNamespace().c_str());
  }
  return true;
}

bool FlightPilot::connectUnity() {
  if (!unity_render_ || unity_bridge_ptr_ == nullptr) return false;
  unity_ready_ = unity_bridge_ptr_->connectUnity(scene_id_);
  return unity_ready_;
}

bool FlightPilot::loadParams(void) {
  // load parameters
  quadrotor_common::getParam("main_loop_freq", main_loop_freq_, pnh_);
  quadrotor_common::getParam("unity_render", unity_render_, pnh_);

  return true;
}

}  // namespace flightros
```

10.17 Navigation through waypoints

In this section, we show how to use rpg_quadrotor_control with Flightmare to navigate through waypoints.

10.17.1 Trajectory

Generate trajectory

```
// Define path through gates
std::vector<Eigen::Vector3d> way_points;
way_points.push_back(Eigen::Vector3d(0, 10, 2.5));
way_points.push_back(Eigen::Vector3d(5, 0, 2.5));
way_points.push_back(Eigen::Vector3d(0, -10, 2.5));
way_points.push_back(Eigen::Vector3d(-5, 0, 2.5));

std::size_t num_waypoints = way_points.size();
Eigen::VectorXd segment_times(num_waypoints);
segment_times << 10.0, 10.0, 10.0, 10.0;
Eigen::VectorXd minimization_weights(5);
minimization_weights << 0.0, 1.0, 1.0, 1.0, 1.0;

polynomial_trajectories::PolynomialTrajectorySettings trajectory_settings =
  polynomial_trajectories::PolynomialTrajectorySettings(
    way_points, minimization_weights, 7, 4);

polynomial_trajectories::PolynomialTrajectory trajectory =
  polynomial_trajectories::minimum_snap_trajectories::
    generateMinimumSnapRingTrajectory(segment_times, trajectory_settings,
                                      20.0, 20.0, 6.0);
```

Get point from trajectory

```
manual_timer timer;
timer.start();

while (ros::ok()) {
  timer.stop();

  quadrotor_common::TrajectoryPoint desired_pose =
    polynomial_trajectories::getPointFromTrajectory(
      trajectory, ros::Duration(timer.get() / 1000));

  // Set pose
  quad_state_.x[QS::POSX] = (Scalar)desired_pose.position.x();
  quad_state_.x[QS::POSY] = (Scalar)desired_pose.position.y();
  quad_state_.x[QS::POSZ] = (Scalar)desired_pose.position.z();
  quad_state_.x[QS::ATTW] = (Scalar)desired_pose.orientation.w();
  quad_state_.x[QS::ATTX] = (Scalar)desired_pose.orientation.x();
  quad_state_.x[QS::ATTY] = (Scalar)desired_pose.orientation.y();
  quad_state_.x[QS::ATTZ] = (Scalar)desired_pose.orientation.z();
```

(continues on next page)

(continued from previous page)

```
  quad_ptr_->setState(quad_state_);

  unity_bridge_ptr_->getRender(0);
  unity_bridge_ptr_->handleOutput();
}
```

Run example

```
roslaunch flightros racing.launch
```

Here the full code example

10.17.2 Header

```
#pragma once

// ros
#include <cv_bridge/cv_bridge.h>
#include <image_transport/image_transport.h>
#include <ros/ros.h>

// standard libraries
#include <assert.h>
#include <Eigen/Dense>
#include <chrono>
#include <cmath>
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <thread>
#include <vector>

// flightlib
#include "flightlib/bridges/unity_bridge.hpp"
#include "flightlib/bridges/unity_message_types.hpp"
#include "flightlib/common/quad_state.hpp"
#include "flightlib/common/types.hpp"
#include "flightlib/objects/quadrotor.hpp"
#include "flightlib/objects/static_gate.hpp"
#include "flightlib/sensors/rgb_camera.hpp"

// trajectory
#include <polynomial_trajectories/minimum_snap_trajectories.h>
#include <polynomial_trajectories/polynomial_trajectories_common.h>
#include <polynomial_trajectories/polynomial_trajectory.h>
```

(continues on next page)

(continued from previous page)

```
#include <polynomial_trajectories/polynomial_trajectory_settings.h>
#include <quadrotor_common/trajectory_point.h>

using namespace flightlib;

namespace racing {

class manual_timer {
  std::chrono::high_resolution_clock::time_point t0;
  double timestamp{0.0};

 public:
  void start() { t0 = std::chrono::high_resolution_clock::now(); }
  void stop() {
    timestamp = std::chrono::duration<double>(
                  std::chrono::high_resolution_clock::now() - t0)
                  .count() *
                1000.0;
  }
  const double &get() { return timestamp; }
};

// void setupQuad();
bool setUnity(const bool render);
bool connectUnity(void);

// unity quadrotor
std::shared_ptr<Quadrotor> quad_ptr_;
std::shared_ptr<RGBCamera> rgb_camera_;
QuadState quad_state_;

// Flightmare(Unity3D)
std::shared_ptr<UnityBridge> unity_bridge_ptr_;
SceneID scene_id_{UnityScene::WAREHOUSE};
bool unity_ready_{false};
bool unity_render_{true};
RenderMessage_t unity_output_;
uint16_t receive_id_{0};
}  // namespace racing
```

10.17.3 Main

```
#include "flightros/racing/racing.hpp"

bool racing::setUnity(const bool render) {
  unity_render_ = render;
  if (unity_render_ && unity_bridge_ptr_ == nullptr) {
    // create unity bridge
    unity_bridge_ptr_ = UnityBridge::getInstance();
    unity_bridge_ptr_->addQuadrotor(quad_ptr_);
```

(continues on next page)

(continued from previous page)

```
    std::cout << "Unity Bridge is created." << std::endl;
  }
  return true;
}

bool racing::connectUnity() {
  if (!unity_render_ || unity_bridge_ptr_ == nullptr) return false;
  unity_ready_ = unity_bridge_ptr_->connectUnity(scene_id_);
  return unity_ready_;
}


int main(int argc, char *argv[]) {
  // initialize ROS
  ros::init(argc, argv, "flightmare_gates");
  ros::NodeHandle nh("");
  ros::NodeHandle pnh("~");
  ros::Rate(50.0);

  // quad initialization
  racing::quad_ptr_ = std::make_unique<Quadrotor>();
  // add camera
  racing::rgb_camera_ = std::make_unique<RGBCamera>();

  // Flightmare
  Vector<3> B_r_BC(0.0, 0.0, 0.3);
  Matrix<3, 3> R_BC = Quaternion(1.0, 0.0, 0.0, 0.0).toRotationMatrix();
  std::cout << R_BC << std::endl;
  racing::rgb_camera_->setFOV(90);
  racing::rgb_camera_->setWidth(720);
  racing::rgb_camera_->setHeight(480);
  racing::rgb_camera_->setRelPose(B_r_BC, R_BC);
  racing::rgb_camera_->setPostProcesscing(std::vector<bool>{
    false, false, false});  // depth, segmentation, optical flow
  racing::quad_ptr_->addRGBCamera(racing::rgb_camera_);

  // // initialization
  racing::quad_state_.setZero();
  racing::quad_ptr_->reset(racing::quad_state_);

  // Initialize gates
  std::string object_id = "unity_gate";
  std::string prefab_id = "rpg_gate";
  std::shared_ptr<StaticGate> gate_1 =
    std::make_shared<StaticGate>(object_id, prefab_id);
  gate_1->setPosition(Eigen::Vector3f(0, 10, 2.5));
  gate_1->setRotation(
    Quaternion(std::cos(0.5 * M_PI_2), 0.0, 0.0, std::sin(0.5 * M_PI_2)));

  std::string object_id_2 = "unity_gate_2";
  std::shared_ptr<StaticGate> gate_2 =
    std::make_unique<StaticGate>(object_id_2, prefab_id);
```

(continues on next page)

(continued from previous page)

```
gate_2->setPosition(Eigen::Vector3f(0, -10, 2.5));
gate_2->setRotation(
  Quaternion(std::cos(0.5 * M_PI_2), 0.0, 0.0, std::sin(0.5 * M_PI_2)));

// Set unity bridge
racing::setUnity(racing::unity_render_);

// Add gates
racing::unity_bridge_ptr_->addStaticObject(gate_1);
racing::unity_bridge_ptr_->addStaticObject(gate_2);

// connect unity
racing::connectUnity();

// Define path through gates
std::vector<Eigen::Vector3d> way_points;
way_points.push_back(Eigen::Vector3d(0, 10, 2.5));
way_points.push_back(Eigen::Vector3d(5, 0, 2.5));
way_points.push_back(Eigen::Vector3d(0, -10, 2.5));
way_points.push_back(Eigen::Vector3d(-5, 0, 2.5));

std::size_t num_waypoints = way_points.size();
Eigen::VectorXd segment_times(num_waypoints);
segment_times << 10.0, 10.0, 10.0, 10.0;
Eigen::VectorXd minimization_weights(5);
minimization_weights << 0.0, 1.0, 1.0, 1.0, 1.0;

polynomial_trajectories::PolynomialTrajectorySettings trajectory_settings =
  polynomial_trajectories::PolynomialTrajectorySettings(
    way_points, minimization_weights, 7, 4);

polynomial_trajectories::PolynomialTrajectory trajectory =
  polynomial_trajectories::minimum_snap_trajectories::
    generateMinimumSnapRingTrajectory(segment_times, trajectory_settings,
                                      20.0, 20.0, 6.0);

// Start racing
racing::manual_timer timer;
timer.start();

while (ros::ok() && racing::unity_render_ && racing::unity_ready_) {
  timer.stop();

  quadrotor_common::TrajectoryPoint desired_pose =
    polynomial_trajectories::getPointFromTrajectory(
      trajectory, ros::Duration(timer.get() / 1000));
  racing::quad_state_.x[QS::POSX] = (Scalar)desired_pose.position.x();
  racing::quad_state_.x[QS::POSY] = (Scalar)desired_pose.position.y();
  racing::quad_state_.x[QS::POSZ] = (Scalar)desired_pose.position.z();
  racing::quad_state_.x[QS::ATTW] = (Scalar)desired_pose.orientation.w();
  racing::quad_state_.x[QS::ATTX] = (Scalar)desired_pose.orientation.x();
  racing::quad_state_.x[QS::ATTY] = (Scalar)desired_pose.orientation.y();
```

(continues on next page)

(continued from previous page)

```
    racing::quad_state_.x[QS::ATTZ] = (Scalar)desired_pose.orientation.z();

    racing::quad_ptr_->setState(racing::quad_state_);

    racing::unity_bridge_ptr_->getRender(0);
    racing::unity_bridge_ptr_->handleOutput();
  }

  return 0;
}
```

10.18 Retrieve simulation data

Note: Currently, all image data are retrieved as CV_8UC3. We are aware of the information loss that occurs and we try to change it ASAP. You can also contribute by fixing this on the server- and client-side.

10.18.1 Setup and spawn the camera

```
// Initialization
std::shared_ptr<Quadrotor> quad_ptr_ = std::make_unique<Quadrotor>();
std::shared_ptr<RGBCamera> rgb_camera_ = std::make_unique<RGBCamera>();

// Setup Camera
Vector<3> B_r_BC(0.0, 0.0, 0.3);
Matrix<3, 3> R_BC = Quaternion(1.0, 0.0, 0.0, 0.0).toRotationMatrix();
rgb_camera_->setFOV(90);
rgb_camera_->setWidth(720);
rgb_camera_->setHeight(480);
rgb_camera_->setRelPose(B_r_BC, R_BC);
rgb_camera_->setPostProcesscing(
  std::vector<bool>{false, false, false});  // depth, segmentation, optical flow
quad_ptr_->addRGBCamera(rgb_camera_);

// Setup Quad
QuadState quad_state_;
quad_state_.setZero();
quad_ptr_->reset(quad_state_);

// Spawn
std::shared_ptr<UnityBridge> unity_bridge_ptr_ = UnityBridge::getInstance();
unity_bridge_ptr_->addQuadrotor(quad_ptr_);
bool unity_ready_ = unity_bridge_ptr_->connectUnity(UnityScene::WAREHOUSE);
```

10.18.2 Position camera

```
// Define new quadrotor state
quad_state_.x[QS::POSX] = (Scalar)position.x;
quad_state_.x[QS::POSY] = (Scalar)position.y;
quad_state_.x[QS::POSZ] = (Scalar)position.z;
quad_state_.x[QS::ATTW] = (Scalar)orientation.w;
quad_state_.x[QS::ATTX] = (Scalar)orientation.x;
quad_state_.x[QS::ATTY] = (Scalar)orientation.y;
quad_state_.x[QS::ATTZ] = (Scalar)orientation.z;

// Set new state
quad_ptr_->setState(quad_state_);
```

10.18.3 Retrieve data

```
// Render next frame
unity_bridge_ptr_->getRender(0);
unity_bridge_ptr_->handleOutput();

cv::Mat img;
rgb_camera_->getRGBImage(img);

// Save image
cv::imwrite("some.jpg", img);
```

10.18.4 [Optional] Publishing data

```
// initialize ROS
ros::init(argc, argv, "flightmare_rviz");
ros::NodeHandle pnh("~");
ros::Rate(50.0);

// initialize publishers
image_transport::ImageTransport it(pnh);
image_transport::Publisher rgb_pub_  = it.advertise("/rgb", 1);

sensor_msgs::ImagePtr rgb_msg =
    cv_bridge::CvImage(std_msgs::Header(), "bgr8", img).toImageMsg();
  rgb_msg->header.stamp.fromNSec(0);

rgb_pub_.publish(rgb_msg);
```

10.18.5 Run example

```
roslaunch flightros camera.launch
```

10.18.6 Here the full code example

Header

```
#pragma once
// ros
#include <cv_bridge/cv_bridge.h>
#include <image_transport/image_transport.h>
#include <ros/ros.h>

// standard libraries
#include <assert.h>
#include <Eigen/Dense>
#include <cmath>
#include <cstring>
#include <experimental/filesystem>
#include <fstream>
#include <iostream>
#include <iterator>
#include <sstream>
#include <thread>
#include <vector>

// flightlib
#include "flightlib/bridges/unity_bridge.hpp"
#include "flightlib/bridges/unity_message_types.hpp"
#include "flightlib/common/quad_state.hpp"
#include "flightlib/common/types.hpp"
#include "flightlib/objects/quadrotor.hpp"
#include "flightlib/sensors/rgb_camera.hpp"

using namespace flightlib;

namespace camera {

// publisher
image_transport::Publisher rgb_pub_;
image_transport::Publisher depth_pub_;
image_transport::Publisher segmentation_pub_;
image_transport::Publisher opticalflow_pub_;
int counter = 0;

// void setupQuad();
bool setUnity(const bool render);
bool connectUnity(void);

// unity quadrotor
```

(continues on next page)

(continued from previous page)

```
std::shared_ptr<Quadrotor> quad_ptr_;
std::shared_ptr<RGBCamera> rgb_camera_;
QuadState quad_state_;

// Flightmare(Unity3D)
std::shared_ptr<UnityBridge> unity_bridge_ptr_;
SceneID scene_id_{UnityScene::WAREHOUSE};
bool unity_ready_{false};
bool unity_render_{true};
RenderMessage_t unity_output_;
uint16_t receive_id_{0};
}  // namespace camera
```

Main

```
#include "flightros/camera/camera.hpp"

bool camera::setUnity(const bool render) {
  unity_render_ = render;
  if (unity_render_ && unity_bridge_ptr_ == nullptr) {
    // create unity bridge
    unity_bridge_ptr_ = UnityBridge::getInstance();
    unity_bridge_ptr_->addQuadrotor(quad_ptr_);
    std::cout << "Unity Bridge is created." << std::endl;
  }
  return true;
}

bool camera::connectUnity() {
  if (!unity_render_ || unity_bridge_ptr_ == nullptr) return false;
  unity_ready_ = unity_bridge_ptr_->connectUnity(scene_id_);
  return unity_ready_;
}


int main(int argc, char *argv[]) {
  // initialize ROS
  ros::init(argc, argv, "flightmare_rviz");
  ros::NodeHandle nh("");
  ros::NodeHandle pnh("~");
  ros::Rate(50.0);

  // initialize publishers
  image_transport::ImageTransport it(pnh);
  camera::rgb_pub_ = it.advertise("/rgb", 1);
  camera::depth_pub_ = it.advertise("/depth", 1);
  camera::segmentation_pub_ = it.advertise("/segmentation", 1);
  camera::opticalflow_pub_ = it.advertise("/opticalflow", 1);

  // quad initialization
```

(continues on next page)

(continued from previous page)

```
camera::quad_ptr_ = std::make_unique<Quadrotor>();
// add mono camera
camera::rgb_camera_ = std::make_unique<RGBCamera>();

// Flightmare
Vector<3> B_r_BC(0.0, 0.0, 0.3);
Matrix<3, 3> R_BC = Quaternion(1.0, 0.0, 0.0, 0.0).toRotationMatrix();
std::cout << R_BC << std::endl;
camera::rgb_camera_->setFOV(90);
camera::rgb_camera_->setWidth(720);
camera::rgb_camera_->setHeight(480);
camera::rgb_camera_->setRelPose(B_r_BC, R_BC);
camera::rgb_camera_->setPostProcesscing(
  std::vector<bool>{true, true, true});  // depth, segmentation, optical flow
camera::quad_ptr_->addRGBCamera(camera::rgb_camera_);


// // initialization
camera::quad_state_.setZero();

camera::quad_ptr_->reset(camera::quad_state_);

// connect unity
camera::setUnity(camera::unity_render_);
camera::connectUnity();

while (ros::ok() && camera::unity_render_ && camera::unity_ready_) {
  camera::quad_state_.x[QS::POSX] = (Scalar)0;
  camera::quad_state_.x[QS::POSY] = (Scalar)0;
  camera::quad_state_.x[QS::POSZ] = (Scalar)0;
  camera::quad_state_.x[QS::ATTW] = (Scalar)0;
  camera::quad_state_.x[QS::ATTX] = (Scalar)0;
  camera::quad_state_.x[QS::ATTY] = (Scalar)0;
  camera::quad_state_.x[QS::ATTZ] = (Scalar)0;

  camera::quad_ptr_->setState(camera::quad_state_);

  camera::unity_bridge_ptr_->getRender(0);
  camera::unity_bridge_ptr_->handleOutput();

  cv::Mat img;

  camera::rgb_camera_->getRGBImage(img);
  sensor_msgs::ImagePtr rgb_msg =
    cv_bridge::CvImage(std_msgs::Header(), "bgr8", img).toImageMsg();
  rgb_msg->header.stamp.fromNSec(camera::counter);
  camera::rgb_pub_.publish(rgb_msg);

  camera::rgb_camera_->getDepthMap(img);
  sensor_msgs::ImagePtr depth_msg =
    cv_bridge::CvImage(std_msgs::Header(), "bgr8", img).toImageMsg();
  depth_msg->header.stamp.fromNSec(camera::counter);
```

(continues on next page)

(continued from previous page)

```
    camera::depth_pub_.publish(depth_msg);

    camera::rgb_camera_->getSegmentation(img);
    sensor_msgs::ImagePtr segmentation_msg =
      cv_bridge::CvImage(std_msgs::Header(), "bgr8", img).toImageMsg();
    segmentation_msg->header.stamp.fromNSec(camera::counter);
    camera::segmentation_pub_.publish(segmentation_msg);

    camera::rgb_camera_->getOpticalFlow(img);
    sensor_msgs::ImagePtr opticflow_msg =
      cv_bridge::CvImage(std_msgs::Header(), "bgr8", img).toImageMsg();
    opticflow_msg->header.stamp.fromNSec(camera::counter);
    camera::opticalflow_pub_.publish(opticflow_msg);

    camera::counter++;
  }

  return 0;
}
```

10.19 Quadrotor References

class Quadrotor

Quadrotor(*const std::string& cfg_path*)

Construct quadrotor.

Parameters
cfg_path (*const std::string&*) – path to configuration yaml file

Return type
Quadrotor

Quadrotor(*const QuadrotorDynamics& dynamics = QuadrotorDynamics(1.0, 0.25)*)

Construct quadrotor.

Parameters
dynamics (*const QuadrotorDynamics& Default=(1.0, 0.25)*) – quadrotor dynamics parameters

Return type
Quadrotor

~Quadrotor();

Deconstruct quadrotor.

Return type
None

reset();

Reset quadrotor's quadstate.

Return type
bool

`reset`(*const QuadState& state*)

Reset quadrotor's quadstate to given quadstate.

Parameters
`state` (*const QuadrotorDynamics&*) – quadrotor new QuadState

Return type
bool

`init();`

Initialize quadrotor's quadstate to default.

Return type
None

`run`(*const Scalar dt*)

Simulate dynamics for dt.

Parameters
`dt` (*const Scalar*) – delta time

Return type
bool

`run`(*const Command& cmd*, *const Scalar dt*)

Simulate dynamics with command for dt.

Parameters

- `cmd` (*const Command&*) – quadrotor command
- `dt` (*const Scalar*) – delta time

Return type
bool

`getState`(*QuadState* const state*)

Get the current state of the quadrotor.

Parameters
`state` (*QuadState* const*) – quadrotor state

Return type
bool

`getMotorThrusts`(*Ref<Vector<4>> motor_thrusts*)

Get the current motor thrust of the quadrotor.

Parameters
`motor_thrusts` (*Ref<Vector<4>>*) – Motor thrusts

Return type
bool

`getMotorOmega`(*Ref<Vector<4>> motor_omega*)

Get the current motor omega of the quadrotor.

Parameters
`motor_omega` (*Ref<Vector<4>>*) – Motor omega

Return type
bool

`getDynamics`(*QuadrotorDynamics* const dynamics*)

Get the quadrotor dynamics.

Parameters
dynamics (*QuadrotorDynamics* const*) – quadrotor dynamics

Return type
bool

`getDynamics`()

Get the quadrotor dynamics.

Return type
const QuadrotorDynamics&

`getSize`()

Get the quadrotor size.

Return type
Vector<3>

`getPosition`()

Get the quadrotor position.

Return type
Vector<3>

`getQuaternion`()

Get the quadrotor orientation.

Return type
Quaternion

`getCameras`()

Get all cameras assigned to the quadrotor.

Return type
std::vector<std::shared_ptr<RGBCamera>>

`getCamera`(*const size_t cam_id*, *std::shared_ptr<RGBCamera> camera*)

Get the camera with the given id which is assigned to the quadrotor.

Parameters

- **cam_id** (*size_t*) – camera ID
- **camera** (*std::shared_ptr<RGBCamera>*) – pointer on camera object

Return type
bool

`setState`(*const QuadState& state*)

Set quadrotor state.

Parameters
state (*const QuadState&*) – quadrotor state

Return type
bool

`setCommand`(*const Command& cmd*)

Set quadrotor command.

Parameters
cmd (*const Command&*) – quadrotor command

Return type
bool

`updateDynamics`(*const QuadrotorDynamics& dynamics*)

Update quadrotor dynamics.

param dynamics
quadrotor dynamics

type dynamics
const QuadrotorDynamics&

rtype
bool

`addRGBCamera`(*std::shared_ptr<RGBCamera> camera*)

Add RGBCamera to quadrotor.

Parameters
camera (*std::shared_ptr<RGBCamera>*) – pointer on camera

Return type
bool

`runFlightCtl`(*const Scalar sim_dt*, *const Vector<3>& omega*, *const Command& cmd*)

Run low-level controller.

Parameters

- **sim_dt** (*const Scalar*) – simulation delta time
- **omega** (*const Vector<3>&*) – omega
- **cmd** (*const Command&*) – quadrotor command

Return type
Vector<4>

`runMotors`(*const Scalar sim_dt*, *const Vector<4>& motor_thrust_des*)

Simulate motor.

Parameters

- **sim_dt** (*const Scalar*) – simulation delta time
- **motor_thrust_des** (*const Vector<4>&*) – desired motor thrust

Return type
None

`setWorldBox`(*const Ref<Matrix<3, 2>> box*)

Set constraints for world.

Parameters
box (*const Ref<Matrix<3, 2>>*) – boundary box

Return type
bool

`constrainInWorldBox`(*const QuadState& old_state*)

Check if quadstate is within the constraints of the world.

Parameters
box (*const QuadState&*) – old_state

Return type
bool

`getMass`()

Get quadrotor mass.

Return type
Scalar

`setSize`(*const Ref<Vector<3>> size*)

Set quadrotor size.

Parameters
size (*const Ref<Vector<3>>*) – quadrotor size

Return type
None

`setCollision`(*const bool collision*)

Set information about quadrotor collision.

Parameters
collision (*const bool*) – if collided

Return type
None

Note: Need to implement getCollision

`getCollision`()

Get information about quadrotor collision.

Return type
bool

10.20 Camera References

`class RGBCamera`

`RGBCamera`()

Construct the RGBCamera.

Return type
RGBCamera

`~RGBCamera()`

Deconstruct the RGBCamera.

Return type
None

`setRelPose`(*const Ref<Vector<3>> B_r_BC*, *const Ref<Matrix<3, 3>> R_BC*)

Set the relative position of the camera to quadrotor.

Parameters

- **B_r_BC** (*const Ref<Vector<3>>*) – relative position
- **R_BC** (*const Ref<Matrix<3, 3>>*) – relative orientation

Return type
bool

setWidth(*const int width*)

Set the width of the camera screen.

Parameters
width (*const int*) – screen width

Return type
bool

setHeight(*const int height*)

Set the height of the camera screen.

Parameters
height (*const int*) – screen height

Return type
bool

setFOV(*const Scalar fov*)

Set the field of view of the camera. Together with the height and the width the camera calibration matrix is defined.

Parameters
fov (*const Scalar*) – Field of view

Return type
bool

setDepthScale(*const Scalar depth_scale*)

Set the depth scale. The value needs to be within the range [0.0, 1.0].

Parameters
depth_scale (*const Scalar*) – Depth scale

Return type
bool

setPostProcesscing(*const std::vector<bool>& enabled_layers*)

Enable the post-processing layers.

Parameters
enabled_layers (*const std::vector<bool>&*) – Layers enables (depth, segmentation, optical flow)

Return type
bool

feedImageQueue(*const int image_layer*, *const cv::Mat& image_mat*)

Store OpenCV mat within a queue.

Parameters

- **image_layer** (*const int*) – Layers (rgb=0, depth=1, segmentation=2, optical flow=3)
- **image_mat** (*const cv::Mat&*) – OpenCV Mat (CV_8UC3)

Return type
bool

`getEnabledLayers()`

Get a list of enabled layers. (depth, segmentation, optical flow)

Return type
std::vector<bool> (Default=false, false, false)

`getRelPose()`

Get the relative pose of the camera.

:rtype:Matrix<4, 4>

`getChannels()`

Get the number of channels of the camera.

Return type
int (Default=3)

`getWidth()`

Get the width of the camera screen.

Return type
int (Default=720)

`getHeight()`

Get the height of the camera screen.

Return type
int (Default=480)

`getFOV()`

Get the field of view of the camera.

Return type
Scalar (Default=70.0)

`getDepthScale()`

Get the depth scale of the camera.

Return type
Scalar (Default=0.2)

`getRGBImage(`*cv::Mat& rgb_img*`)`

Get an image from the RGBImageQueue. Mat format CV_8UC3.

Return type
bool, successfully retrieved an image

`getDepthMap(`*cv::Mat& depth_map*`)`

Get an image from the DepthMapQueue. Mat format CV_8UC3.

Return type
bool, successfully retrieved an image

`getSegmentation(`*cv::Mat& segmentation*`)`

Get an image from the SegmentationQueue. Mat format CV_8UC3.

Return type
bool, successfully retrieved an image

`getOpticalFlow`(*cv::Mat& opticalflow*)

Get an image from the OpticalFlowQueue. Mat format CV_8UC3.

Return type
bool, successfully retrieved an image

`enableDepth`(*const bool on*)

Auxiliary function to enable the depth.

Parameters
on (*bool*) – Enable depth

Return type
None

`enableSegmentation`(*const bool on*)

Auxiliary function to enable the segmentation.

Parameters
on (*bool*) – Enable segmentation

Return type
None

`enableOpticalFlow`(*const bool on*)

Auxiliary function to enable the optical flow.

Parameters
on (*bool*) – Enable optical flow

Return type
None

10.21 StaticGate References

`class StaticGate`

`StaticGate`(*std::string id*, *std::string prefab_id*)

Constuct Static gate.

Parameters

- **id** (*std::string*) – unique name
- **prefab_id** (*std::string*) – prefab name in unity

Return type
StaticGate

`~StaticGate()`

Deconstuct Static gate.

Return type
None

`setPosition`(*const Vector<3>& position*)

Set the position of the gate.

Parameters
position (*const Vector<3>&*) – position of the gate

Return type
None

`setRotation`(*const Quaternion& quaternion*)

Set the orientation of the gate.

Parameters
quaternion (*const Quaternion&*) – orientation of the gate

Return type
None

`setSize`(*const Vector<3>& size*)

Set the size of the gate.

Parameters
size (*const Vector<3>&*) – size of the gate

Return type
None

`getPos`()

Get the position of the gate.

Return type
Vector<3>

`getQuat`()

Get the orientation of the gate.

Return type
Quaternion

`getSize`()

Get the size of the gate.

Return type
Vector<3>

10.22 Quadrotor environment reference

`class QuadrotorEnv`

`QuadrotorEnv`()

Constructs a QuadrotorEnv from the configuration file "/flightlib/configs/quadrotor_env.yaml".

Return type
QuadrotorEnv

`QuadrotorEnv`(*const std::string &cfg_path*)

Constructs a QuadrotorEnv from the given configuration file.

Parameters
cfg_path (*std::string*) – configuration file path

Return type
QuadrotorEnv

`~QuadrotorEnv()`

Deconstructs this QuadrotorEnv.

Return type
None

`reset`(*Ref<Vector<>> obs*, *const bool random*)

Reset the quadrotor state and get observations.

//full states of the quadrotor
obs = [position, orientation, linear velocity, angular velocity]

Parameters

- **obs** (*Ref<Vector<>>*) – Observations
- **random** (*const bool*) – randomly reset the quadrotor state

Return type
bool

`getObs`(*Ref<Vector<>> obs*)

Get observations.

//full states of the quadrotor
obs = [position, orientation, linear velocity, angular velocity]

Parameters
obs (*Ref<Vector<>>*) – Observations

Return type
bool

`step`(*const Ref<Vector<>> act*, *Ref<Vector<>> obs*)

Simulate one step with the given thrusts.

Get observations and the total reward of this step.

// thrusts of each propeller
act = [f1, f2, f3, f4]

//full states of the quadrotor
obs = [position, orientation, linear velocity, angular velocity]

Parameters

- **act** (*const Ref<Vector<>>*) – thrusts of each propeller
- **obs** (*Ref<Vector<>>*) – Observations

Return type
Total reward (Scalar)

`isTerminalState`(*Scalar &reward*)

Check if terminal state is reached.

Parameters
reward (*Scalar*) – Reward of state

Return type
bool

`loadParam`(*const YAML::Node &cfg*)

Load the QuadrotorEnv parameters.

Parameters
cfg (*const YAML::Node*) – configuration YAML

Return type
bool

`getAct`(*Ref<Vector<>> act*)

Get thrusts for commands.

// thrusts of each propeller
act = [f1, f2, f3, f4]

Parameters
act (*const Ref<Vector<>>*) – thrusts of each propeller

Return type
bool

`getAct`(*Command *const cmd*)

Get the next command for the quadrotor.

Parameters
cmd (*Command*) – Quadrotor command

Return type
bool

`addObjectsToUnity`(*std::shared_ptr<UnityBridge> bridge*)

Add the quadrotor to Flightmare.

Parameters
bridge (*std::shared_ptr<UnityBridge>*) – Unity bridge

Return type
None

10.23 FlightEnvVec

The wrapper of Flightmare.

class FlightEnvVec(*impl*)

An OpenAIGym like environment of Flightmare.

env = FlightEnvVec.FlightEnvVec(QuadrotorEnv_v1(
dump(cfg, Dumper=RoundTripDumper), False))

param impl
Implementation of flightgym

type impl
flightgym.QuadrotorEnv_v1

seed(*self*, *seed=0*)

Set a random seed for the environment.

Parameters
seed (*int*) – Random seed
Return type
bool

step(*self*, *action*)

Simulate one step with the given thrusts.

Get observations and the total reward of this step.

// thrusts of each propeller
action = [f1, f2, f3, f4]

//full states of the quadrotor
obs = [position, orientation, linear velocity, angular velocity]

Parameters
action (*np.ndarray*) – thrusts of each propeller
Return type
observations, reward, done, info

Error: stepUnity() not yet implemented.

stepUnity(*self*, *action*, *send_id*)

Simulate one step with the given thrusts and render it in Unity.

Get observations and the total reward of this step.

// thrusts of each propeller
action = [f1, f2, f3, f4]

//full states of the quadrotor
obs = [position, orientation, linear velocity, angular velocity]

Parameters
- **action** (*np.ndarray*) – thrusts of each propeller
- **send_id** (*int*) – frame ID

Return type
observations, reward, done, info

sample_actions(*self*)

Sample an action.

// thrusts of each propeller
action = [f1, f2, f3, f4]

Return type
action (np.ndarray)

reset()

Reset the quadrotor state and get observations.

//full states of the quadrotor
obs = [position, orientation, linear velocity, angular velocity]

Return type
obs

reset_and_update_info(*self*)

Reset the quadrotor state, get observations and update info.

//full states of the quadrotor
obs = [position, orientation, linear velocity, angular velocity]

Return type
obs, info

Error: render() not yet implemented.

render(*self*, *mode='human'*)

Render in Flightmare.

rtype
None

close(*self*)

Close the environment.

Return type
None

`connectUnity`(*self*)

Connect to Flightmare.

Return type
None

`disconnectUnity`(*self*)

Disconnect from Flightmare.

Return type
None

INDEX

Q

R

S