airsim Documentation

Shital Shah

Nov 09, 2018

Contents

AirSim is a simulator for drones, cars and more, built on Unreal Engine. It is open-source, cross platform and supports hardware-in-loop with popular flight controllers such as PX4 for physically and visually realistic simulations. It is developed as an Unreal plugin that can simply be dropped into any Unreal environment you want.

Our goal is to develop AirSim as a platform for AI research to experiment with deep learning, computer vision and reinforcement learning algorithms for autonomous vehicles. For this purpose, AirSim also exposes APIs to retrieve data and control vehicles in a platform independent way.

AirSim Drone Demo Video

AirSim Car Demo Video

CHAPTER 1

Installing

1.1 Windows

- Binaries
- Build from source

1.2 Linux

- Build from source

CHAPTER 2

Getting started

2.1 Choosing the Mode: Car, Multirotor or ComputerVision

By default AirSim will prompt you to choose Car or Multirotor mode. You can use SimMode setting to specify the default vehicle or the new ComputerVision mode.

2.2 Manual drive

If you have remote control (RC) as shown below, you can manually control the drone in the simulator. For cars, you can use arrow keys to drive manually.

More details

Fig. 1: record screenshot

Fig. 2: record screenshot

2.3 Programmatic control

AirSim exposes APIs so you can interact with the vehicle in the simulation programmatically. You can use these APIs to retrieve images, get state, control the

- Reference
- API
- Settings
- FAQ

2.3.1 Using C++ APIs for AirSim

Please read general API doc first if you haven't already. This document describes C++ examples and other C++ specific details.

Quick Start

Fastest way to get started is to open AirSim.sln in Visual Studio 2017. You will see Hello Car and Hello Drone examples in the solution. These examples will show you the include paths and lib paths you will need to setup in your VC++ projects. If you are using Linux then you will specify these paths either in your cmake file or on compiler command line.

Include and Lib Folders

- Include folders: `$(ProjectDir)..\AirLib\deps\rpclib\include;include;$(ProjectDir)..\AirLib\deps\eigen3;$(ProjectDir)..\AirLib\include`
- Dependencies: `rpc.lib`
- Lib folders: `$(ProjectDir)\..\AirLib\deps\MavLinkCom\lib\$(Platform)\$(Configuration);$(ProjectDir)\..\AirLib\deps\rpclib\lib\$(Platform)\$(Configuration);$(ProjectDir)\..\AirLib\lib\$(Platform)\$(Configuration)`

Hello Car

Here's how to use AirSim APIs using Python to control simulated car (see also Python example):

```
// ready to run example: https://github.com/Microsoft/AirSim/blob/master/HelloCar/
↪main.cpp

#include <iostream>
#include "vehicles/car/api/CarRpcLibClient.hpp"

int main()
{
    msr::airlib::CarRpcLibClient client;
    client.enableApiControl(true); //this disables manual control
    CarControllerBase::CarControls controls;

    std::cout << "Press enter to drive forward" << std::endl; std::cin.get();
    controls.throttle = 1;
    client.setCarControls(controls);

    std::cout << "Press Enter to activate handbrake" << std::endl; std::cin.get();
    controls.handbrake = true;
    client.setCarControls(controls);

    std::cout << "Press Enter to take turn and drive backward" << std::endl; std::cin.
↪get();
    controls.handbrake = false;
    controls.throttle = -1;
    controls.steering = 1;
    client.setCarControls(controls);
```

(continues on next page)

(continued from previous page)

```
    std::cout << "Press Enter to stop" << std::endl; std::cin.get();
    client.setCarControls(CarControllerBase::CarControls());

    return 0;
}
```

Hello Drone

Here's how to use AirSim APIs using Python to control simulated car (see also Python example):

```
// ready to run example: https://github.com/Microsoft/AirSim/blob/master/HelloDrone/
↪main.cpp

#include <iostream>
#include "vehicles/multirotor/api/MultirotorRpcLibClient.hpp"

int main()
{
    using namespace std;
    msr::airlib::MultirotorRpcLibClient client;

    cout << "Press Enter to enable API control" << endl; cin.get();
    client.enableApiControl(true);

    cout << "Press Enter to arm the drone" << endl; cin.get();
    client.armDisarm(true);

    cout << "Press Enter to takeoff" << endl; cin.get();
    client.takeoffAsync(5)->waitOnLastTask();

    cout << "Press Enter to move 5 meters in x direction with 1 m/s velocity" << endl;
↪ cin.get();
    auto position = client.getPosition(); // from current location
    client.moveToPositionAsync(position.x() + 5, position.y(), position.z(), 1)->
↪waitOnLastTask();

    cout << "Press Enter to land" << endl; cin.get();
    client.landAync()->waitOnLastTask();

    return 0;
}
```

See Also

- Examples of how to use internal infrastructure in AirSim in your other projects
- DroneShell app shows how to make simple interface using C++ APIs to control drones
- Python APIs

2.3.2 AirSim APIs

Introduction

AirSim exposes APIs so you can interact with vehicle in the simulation programmatically. You can use these APIs to retrieve images, get state, control the vehicle and so on.

Python Quickstart

If you want to use Python to call AirSim APIs, we recommend using Anaconda with Python 3.5 or later versions however some code may also work with Python 2.7 (help us improve compatibility!).

First install this package:

You can either get AirSim binaries from releases or compile from the source (Windows, Linux). Once you can run AirSim, choose Car as vehicle and then navigate to `PythonClient\car\` folder and run:

If you are using Visual Studio 2017 then just open AirSim.sln, set PythonClient as startup project and choose `car\hello_car.py` as your startup script.

Installing AirSim Package

You can also install `airsim` package simply by,

You can find source code and samples for this package in `PythonClient` folder in your repo.

Notes

1. You may notice a file `setup_path.py` in our example folders. This file has simple code to detect if `airsim` package is available in parent folder and in that case we use that instead of pip installed package so you always use latest code.
2. AirSim is still under heavy development which means you might frequently need to update the package to use new APIs.

C++ Users

If you want to use C++ APIs and examples, please see C++ APIs Guide.

Hello Car

Here's how to use AirSim APIs using Python to control simulated car (see also C++ example):

```
# ready to run example: PythonClient/car/hello_car.py
import airsim
import time

# connect to the AirSim simulator
client = airsim.CarClient()
client.confirmConnection()
client.enableApiControl(True)
car_controls = airsim.CarControls()

while True:
```

(continues on next page)

(continued from previous page)

```
    # get state of the car
    car_state = client.getCarState()
    print("Speed %d, Gear %d" % (car_state.speed, car_state.gear))

    # set the controls for car
    car_controls.throttle = 1
    car_controls.steering = 1
    client.setCarControls(car_controls)

    # let car drive a bit
    time.sleep(1)

    # get camera images from the car
    responses = client.simGetImages([
        airsim.ImageRequest(0, airsim.ImageType.DepthVis),
        airsim.ImageRequest(1, airsim.ImageType.DepthPlanner, True)])
    print('Retrieved images: %d', len(responses))

    # do something with images
    for response in responses:
        if response.pixels_as_float:
            print("Type %d, size %d" % (response.image_type, len(response.image_data_
↪float)))
            airsim.write_pfm('py1.pfm', airsim.get_pfm_array(response))
        else:
            print("Type %d, size %d" % (response.image_type, len(response.image_data_
↪uint8)))
            airsim.write_file('py1.png', response.image_data_uint8)
```

Hello Drone

Here's how to use AirSim APIs using Python to control simulated quadrotor (see also C++ example):

```
# ready to run example: PythonClient/multirotor/hello_drone.py
import airsim

# connect to the AirSim simulator
client = airsim.MultirotorClient()
client.confirmConnection()
client.enableApiControl(True)
client.armDisarm(True)

# Async methods returns Future. Call join() to wait for task to complete.
client.takeoffAsync().join()
client.moveToPositionAsync(-10, 10, -10, 5).join()

# take images
responses = client.simGetImages([
    airsim.ImageRequest("0", airsim.ImageType.DepthVis),
    airsim.ImageRequest("1", airsim.ImageType.DepthPlanner, True)])
print('Retrieved images: %d', len(responses))

# do something with the images
for response in responses:
    if response.pixels_as_float:
```

(continues on next page)

(continued from previous page)

```
        print("Type %d, size %d" % (response.image_type, len(response.image_data_
↪float)))
        airsim.write_pfm(os.path.normpath('/temp/py1.pfm'), airsim.
↪getPfmArray(response))
    else:
        print("Type %d, size %d" % (response.image_type, len(response.image_data_
↪uint8)))
        airsim.write_file(os.path.normpath('/temp/py1.png'), response.image_data_
↪uint8)
```

Common APIs

- `reset`: This resets the vehicle to its original starting state. Note that you must call `enableApiControl` and `armDisarm` again after the call to `reset`.
- `confirmConnection`: Checks state of connection every 1 sec and reports it in Console so user can see the progress for connection.
- `enableApiControl`: For safety reasons, by default API control for autonomous vehicle is not enabled and human operator has full control (usually via RC or joystick in simulator). The client must make this call to request control via API. It is likely that human operator of vehicle might have disallowed API control which would mean that enableApiControl has no effect. This can be checked by `isApiControlEnabled`.
- `isApiControlEnabled`: Returns true if API control is established. If false (which is default) then API calls would be ignored. After a successful call to `enableApiControl`, the `isApiControlEnabled` should return true.
- `ping`: If connection is established then this call will return true otherwise it will be blocked until timeout.
- *simPrintLogMessage*: Prints the specified message in the simulator's window. If message_param is also supplied then its printed next to the message and in that case if this API is called with same message value but different message_param again then previous line is overwritten with new line (instead of API creating new line on display). For example, *simPrintLogMessage("Iteration: ", to_string(i))* keeps updating same line on display when API is called with different values of i. The valid values of severity parameter is 0 to 3 inclusive that corresponds to different colors.
- `simGetObjectPose`, *simSetObjectPose*: Gets and sets the pose of specified object in Unreal environment. Here the object means "actor" in Unreal terminology. They are searched by tag as well as name. Please note that the names shown in UE Editor are *auto-generated* in each run and are not permanent. So if you want to refer to actor by name, you must change its auto-generated name in UE Editor. Alternatively you can add a tag to actor which can be done by clicking on that actor in Unreal Editor and then going to [Tags property](https://answers.unrealengine.com/questions/543807/whats-the-difference-between-tag-and-tag.html), click "+" sign and add some string value. If multiple actors have same tag then the first match is returned. If no matches are found then NaN pose is returned. The returned pose is in NED coordinates in SI units with its origin at Player Start. For `simSetObjectPose`, the specified actor must have Mobility set to Movable or otherwise you will get undefined behavior. The `simSetObjectPose` has parameter `teleport` which means object is moved through other objects in its way and it returns true if move was successful

Image / Computer Vision APIs

AirSim offers comprehensive images APIs to retrieve synchronized images from multiple cameras along with ground truth including depth, disparity, surface normals and vision. You can set the resolution, FOV, motion blur etc parameters in settings.json. There is also API for detecting collision state. See also complete code that generates specified number of stereo images and ground truth depth with normalization to camera plan, computation of disparity image and saving it to pfm format.

More on image APIs and Computer Vision mode.

Pause and Continue APIs

AirSim allows to pause and continue the simulation through `pause(is_paused)` API. To pause the simulation call `pause(True)` and to continue the simulation call `pause(False)`. You may have scenario, especially while using reinforcement learning, to run the simulation for specified amount of time and then automatically pause. While simulation is paused, you may then do some expensive computation, send a new command and then again run the simulation for specified amount of time. This can be achieved by API `continueForTime(seconds)`. This API runs the simulation for the specified number of seconds and then pauses the simulation. For example usage, please see pause_continue_car.py and pause_continue_drone.py.

Collision API

The collision information can be obtained using `simGetCollisionInfo` API. This call returns a struct that has information not only whether collision occurred but also collision position, surface normal, penetration depth and so on.

Multiple Vehicles

AirSim supports multiple vehicles and control them through APIs. Please Multiple Vehicles doc.

Coordinate System

All AirSim API uses NED coordinate system, i.e., +X is North, +Y is East and +Z is Down. All units are in SI system. Please note that this is different from coordinate system used internally by Unreal Engine. In Unreal Engine, +Z is up instead of down and length unit is in centimeters instead of meters. AirSim APIs takes care of the appropriate conversions. The starting point of the vehicle is always coordinates (0, 0, 0) in NED system. Thus when converting from Unreal coordinates to NED, we first subtract the starting offset and then scale by 100 for cm to m conversion. The vehicle is spawned in Unreal environment where the Player Start component is placed. There is a setting called `OriginGeopoint` in settings.json which assigns geographic longitude, longitude and altitude to the Player Start component.

Vehicle Specific APIs

APIs for Car

Car has followings APIs available:

- `setCarControls`: This allows you to set throttle, steering, handbrake and auto or manual gear.
- *getCarState*: This retrieves the state information including speed, current gear and 6 kinematics quantities: position, orientation, linear and angular velocity, linear and angular acceleration. All quantities are in NED coordinate system, SI units in world frame except for angular velocity and accelerations which are in body frame.
- Image APIs.

APIs for Multirotor

Multirotor can be controlled by specifying angles, velocity vector, destination position or some combination of these. There are corresponding `move*` APIs for this purpose. When doing position control, we need to use some path following algorithm. By default AirSim uses carrot following algorithm. This is often referred to as "high level control" because you just need to specify high level goal and the firmware takes care of the rest. Currently lowest level control available in AirSim is `moveByAngleThrottleAsync` API.

getMultirotorState

This API returns the state of the vehicle in one call. The state includes, collision, estimated kinematics (i.e. kinematics computed by fusing sensors), and timestamp (nano seconds since epoch). The kinematics here means 6 quantities: position, orientation, linear and angular velocity, linear and angular acceleration. Please note that simple_slight currently doesn't support state estimator which means estimated and ground truth kinematics values would be same for simple_flight. Estimated kinematics are however available for PX4 except for angular acceleration. All quantities are in NED coordinate system, SI units in world frame except for angular velocity and accelerations which are in body frame.

Async methods, duration and max_wait_seconds

Many API methods has parameters named `duration` or `max_wait_seconds` and they have *Async* as suffix, for example, `takeoffAsync`. These methods will return immediately after starting the task in AirSim so that your client code can do something else while that task is being executed. If you want to wait for this task to complete then you can call `waitOnLastTask` like this:

```
//C++
client.takeoffAsync()->waitOnLastTask();
```

```
# Python
client.takeoffAsync().join()
```

If you start another command then it automatically cancels the previous task and starts new command. This allows to use pattern where your coded continuously does the sensing, computes a new trajectory to follow and issues that path to vehicle in AirSim. Each newly issued trajectory cancels the previous trajectory allowing your code to continuously do the update as new sensor data arrives.

All *Async* method returns `concurrent.futures.Future` in Python (`std::future` in C++). Please note that these future classes currently do not allow to check status or cancel the task; they only allow to wait for task to complete. AirSim does provide API `cancelLastTask`, however.

drivetrain

There are two modes you can fly vehicle: `drivetrain` parameter is set to `airsim.Drivetrain.ForwardOnly` or `airsim.Drivetrain.MaxDegreeOfFreedom`. When you specify ForwardOnly, you are saying that vehicle's front should always point in the direction of travel. So if you want drone to take left turn then it would first rotate so front points to left. This mode is useful when you have only front camera and you are operating vehicle using FPV view. This is more or less like travelling in car where you always have front view. The MaxDegreeOfFreedom means you don't care where the front points to. So when you take left turn, you just start going left like crab. Quadrotors can go in any direction regardless of where front points to. The MaxDegreeOfFreedom enables this mode.

yaw_mode

`yaw_mode` is a struct `YawMode` with two fields, `yaw_or_rate` and `is_rate`. If `is_rate` field is True then `yaw_or_rate` field is interpreted as angular velocity in degrees/sec which means you want vehicle to rotate continuously around its axis at that angular velocity while moving. If `is_rate` is False then `yaw_or_rate` is interpreted as angle in degrees which means you want vehicle to rotate to specific angle (i.e. yaw) and keep that angle while moving.

You can probably see that when `yaw_mode.is_rate == true`, the `drivetrain` parameter shouldn't be set to `ForwardOnly` because you are contradicting by saying that keep front pointing ahead but also rotate continuously. However if you have `yaw_mode.is_rate = false` in `ForwardOnly` mode then you can do some funky stuff. For example, you can have drone do circles and have yaw_or_rate set to 90 so camera is always pointed to center ("super cool selfie mode"). In `MaxDegreeofFreedom` also you can get some funky stuff by setting `yaw_mode.is_rate = true` and say `yaw_mode.yaw_or_rate = 20`. This will cause drone to go in its path while rotating which may allow to do 360 scanning.

In most cases, you just don't want yaw to change which you can do by setting yaw rate of 0. The shorthand for this is `airsim.YawMode.Zero()` (or in C++: `YawMode::Zero()`).

lookahead and adaptive_lookahead

When you ask vehicle to follow a path, AirSim uses "carrot following" algorithm. This algorithm operates by looking ahead on path and adjusting its velocity vector. The parameters for this algorithm is specified by `lookahead` and `adaptive_lookahead`. For most of the time you want algorithm to auto-decide the values by simply setting `lookahead = -1` and `adaptive_lookahead = 0`.

Using APIs on Real Vehicles

We want to be able to run *same code* that runs in simulation as on real vehicle. This allows you to test your code in simulator and deploy to real vehicle.

Generally speaking, APIs therefore shouldn't allow you to do something that cannot be done on real vehicle (for example, getting the ground truth). But, of course, simulator has much more information and it would be useful in applications that may not care about running things on real vehicle. For this reason, we clearly delineate between sim-only APIs by attaching `sim` prefix, for example, `simGetGroundTruthKinematics`. This way you can avoid using these simulation-only APIs if you care about running your code on real vehicles.

The AirLib is self-contained library that you can put on an offboard computing module such as the Gigabyte barebone Mini PC. This module then can talk to the flight controllers such as PX4 using exact same code and flight controller protocol. The code you write for testing in the simulator remains unchanged. See AirLib on custom drones.

Adding New APIs to AirSim

Adding new APIs requires modifying the source code. Much of the changes are mechanical and required for various levels of abstractions that AirSim supports. This commit demonstrates how to add a simple API `simPrintLogMessage` that prints message in simulator window.

Some Internals

The APIs use msgpack-rpc protocol over TCP/IP through rpclib developed by Tamás Szelei which allows you to use variety of programming languages including C++, C#, Python, Java etc. When AirSim starts, it opens port 41451 (this can be changed via settings) and listens for incoming request. The Python or C++ client code connects to this port and sends RPC calls using msgpack serialization format.

References and Examples

- C++ API Examples
- Car Examples
- Multirotor Examples
- Computer Vision Examples
- Move on Path demo showing video of fast multirotor flight through Modular Neighborhood environment
- Building a Hexacopter
- Building Point Clouds

FAQ

If you see Unreal getting slowed down dramatically when Unreal Engine window loses focus then go to 'Edit->Editor Preferences' in Unreal Editor, in the 'Search' box type 'CPU' and ensure that the 'Use Less CPU when in Background' is unchecked.

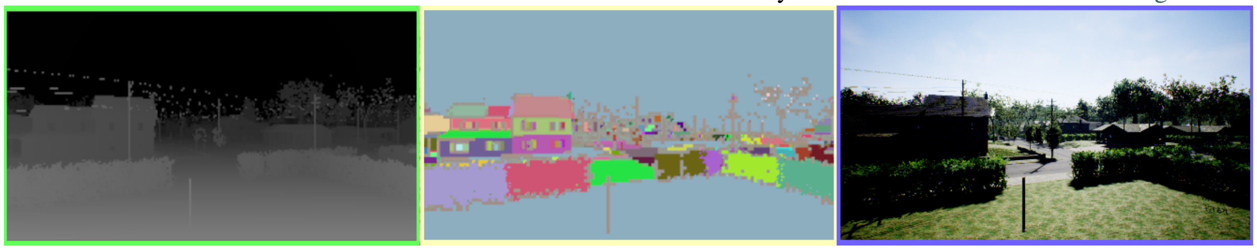
You should install VS2017 with VC++, Windows SDK 8.1 and Python. To use Python APIs you will need Python 3.5 or later (install it using Anaconda).

We recommend Anaconda to get Python tools and libraries. Our code is tested with Python 3.5.3 :: Anaconda 4.4.0. This is important because older version have been known to have problems.

You can install OpenCV using:

2.3.3 Camera Views

The camera views that are shown on screen are the camera views you can fetch via the simGetImages API.



From left to right is the depth view, segmentation view and the FPV view. See Image APIs for description of various available views.

Turning ON/OFF Views

Press F1 key to see keyboard shortcuts for turning on/off any or all views. You can also select various view modes there, such as "Fly with Me" mode, FPV mode and "Ground View" mode.

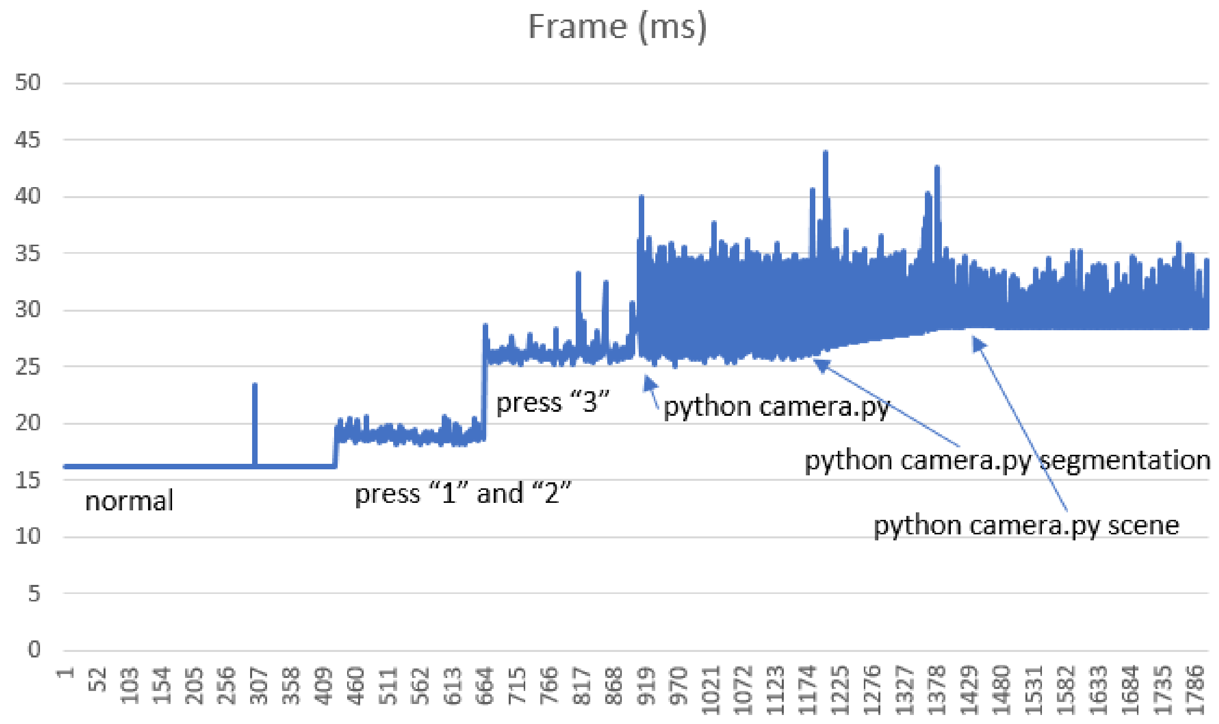
Configuring Sub-Windows

Now you can select what is shown by each of above sub windows. For instance, you can chose to show surface normals in first window (instead of depth) and disparity in second window (instead of segmentation). Below is the settings value you can use in settings.json:

Performance Impact

Note: This section is outdated and has not been updated for new performance enhancement changes.

Now rendering these views does impact the FPS performance of the game, since this is additional work for the GPU. The following shows the impact on FPS when you open these views.

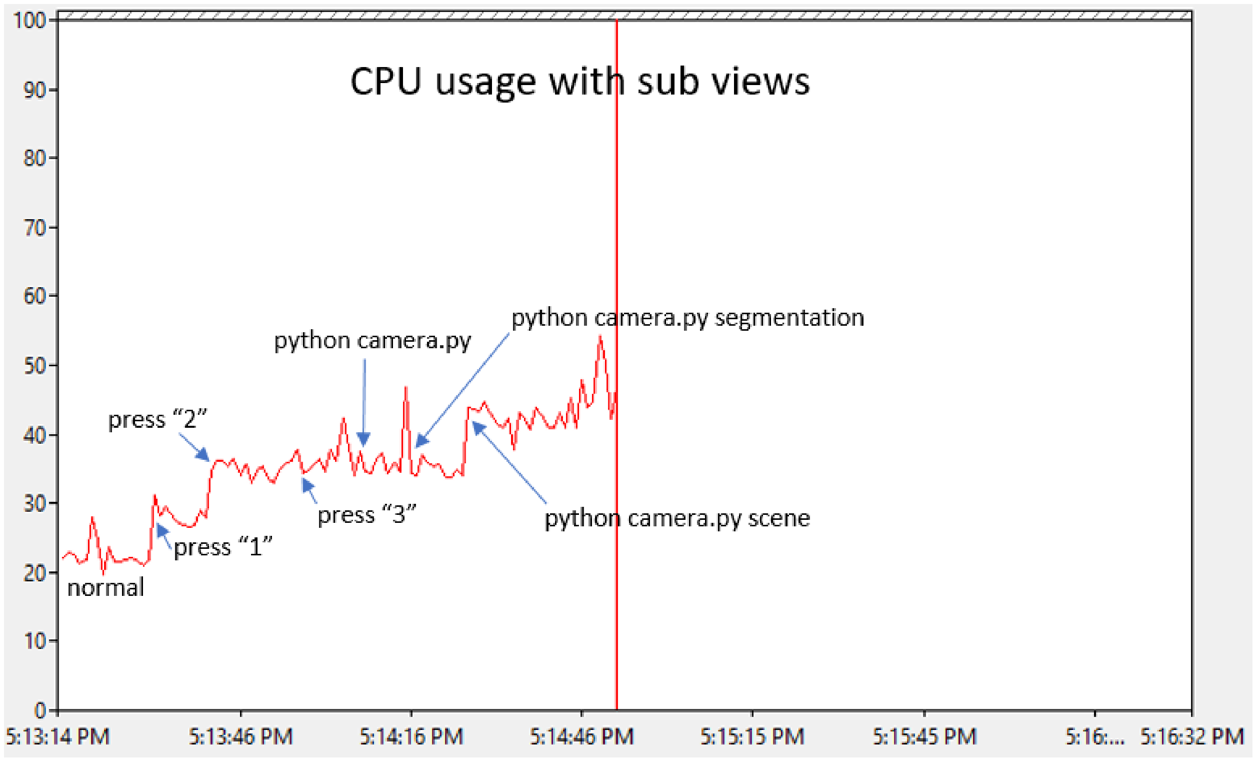


This is measured on Intel core i7 computer with 32 gb RAM and a GeForce GTX 1080 graphics card running the Modular Neighborhood map, using cooked debug bits, no debugger or GameEditor open. The normal state with no subviews open is measuring around 16 ms per frame, which means it is keeping a nice steady 60 FPS (which is the target FPS). As it climbs up to 35ms the FPS drops to around 28 frames per second, spiking to 40ms means a few drops to 25 fps.

The simulator can still function and fly correctly when all this is going on even in the worse case because the physics is decoupled from the rendering. However if the delay gets too high such that the communication with PX4 hardware is interrupted due to overly busy CPU then the flight can stall due to timeout in the offboard control messages.

On the computer where this was measured the drone could fly the path.py program without any problems with all views open, and with 3 python scripts running to capture each view type. But there was one stall during this flight, but it recovered gracefully and completed the path. So it was right on the limit.

The following shows the impact on CPU, perhaps a bit surprisingly, the CPU impact is also non trivial.



2.3.4 Installing cmake on Linux

If you don’t have cmake version 3.10 (for example, 3.2.2 is the default on Ubuntu 14) you can run the following:

Now you have to run this command by itself (it is interactive)

Answer ‘n’ to the question about creating another cmake-3.10.2-Linux-x86_64 folder and then

Now type `cmake --version` to make sure your cmake version is 3.10.2.

2.3.5 AirLib

Majority of the code is located in AirLib. This is a self-contained library that you should be able to compile with any C++11 compiler.

AirLib consists of the following components:

1. *Physics engine:* This is header-only physics engine. It is designed to be fast and extensible to implement different vehicles.
2. *Sensor models:* This is header-only models for Barometer, IMU, GPS and Magnetometer
3. *Vehicle models:* This is header-only models for vehicle configurations and models. Currently we have implemented model for a MultiRotor and a configuration for PX4 QuadRotor in the X config.
4. *Control library:* This part of AirLib provides abstract base class for our APIs and concrete implementation for specific vehicle platforms such as MavLink. It also has classes for the RPC client and server.

2.3.6 Unreal/Plugins/AirSim

This is the only portion of project which is dependent on Unreal engine. We have kept it isolated so we can implement simulator for other platforms as well (for example, Unity). The Unreal code takes advantage of its UObject based classes including Blueprints.

1. *SimMode_ classes*: We wish to support various simulator modes such as pure Computer Vision mode where there is no drone. The SimMode classes help implement many different modes.
2. *VehiclePawnBase*: This is the base class for all vehicle pawn visualizations.
3. *VehicleBase*: This class provides abstract interface to implement a combination of rendering component (i.e. Unreal pawn), physics component (i.e. MultiRotor) and controller (i.e. MavLinkHelper).

2.3.7 MavLinkCom

This is the library developed by our own team member Chris Lovett that provides C++ classes to talk to the MavLink devices. This library is stand alone and can be used in any project. See MavLinkCom for more info.

2.3.8 Sample Programs

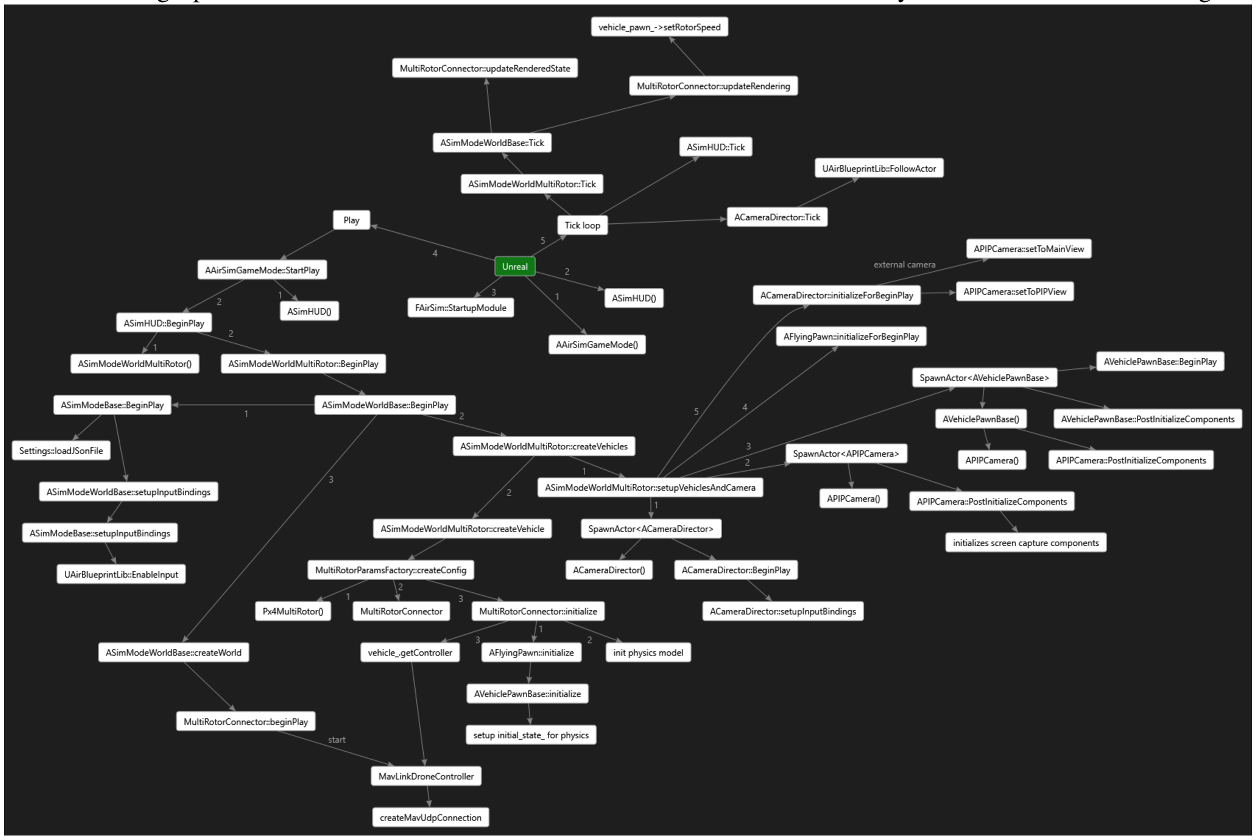
We have created a few sample programs to demonstrate how to use the API. See HelloDrone and DroneShell. DroneShell demonstrates how to connect to the simulator using UDP. The simulator is running a server (similar to DroneServer).

2.3.9 Contributing

See Contribution Guidelines

2.3.10 Unreal Framework

The following picture illustrates how AirSim is loaded and invoked by the Unreal Game Engine:



vehicle_pawn_->setRotorSpeed
MultiRotorConnector::updateRenderedState
MultiRotorConnector::updateRendering
ASimModeWorldBase::Tick
ASimHUD::Tick
UAirBlueprintLib::FollowActor
ASimModeWorldMultiRotor::Tick
Play
Tick loop
ACameraDirector::Tick
APIPCamera::setToMainView
external camera
Unreal
AAirSimGameMode::StartPlay
ACameraDirector::initializeForBeginPlay
APIPCamera::setToPIPView
FAirSim::StartupModule
ASimHUD()
ASimHUD()
ASimHUD::BeginPlay
AFlyingPawn::initializeForBeginPlay
AAirSimGameMode()
ASimModeWorldMultiRotor()
ASimModeWorldMultiRotor::BeginPlay
AVehiclePawnBase::BeginPlay
SpawnActor<AVehiclePawnBase>
ASimModeBase::BeginPlay
ASimModeWorldBase::BeginPlay
AVehiclePawnBase()
AVehiclePawnBase::PostInitializeComponents
Settings::loadJSonFile
ASimModeWorldMultiRotor::createVehicles
SpawnActor<APIPCamera>
APIPCamera()
APIPCamera::PostInitializeComponents
ASimModeWorldBase::setupInputBindings
ASimModeWorldMultiRotor::setupVehiclesAndCamera
APIPCamera()
APIPCamera::PostInitializeComponents
ASimModeBase::setupInputBindings
ASimModeWorldMultiRotor::createVehicle
SpawnActor<ACameraDirector>
initializes screen capture components
UAirBlueprintLib::EnableInput
MultiRotorParamsFactory::createConfig
ACameraDirector()
ACameraDirector::BeginPlay
Px4MultiRotor()
MultiRotorConnector
MultiRotorConnector::initialize
ACameraDirector::setupInputBindings
ASimModeWorldBase::createWorld
vehicle_.getController
AFlyingPawn::initialize
init physics model
AVehiclePawnBase::initialize
MultiRotorConnector::beginPlay
setup initial_state_ for physics
start
MavLinkDroneController
createMavUdpConnection

2.3.11 Modern C++ Coding Guidelines

We are using Modern C++11. Smart pointers, Lambdas, and C++11 multithreading primitives are your friend.

Quick Note

The great thing about "standards" is that there are many to chose from: ISO, Sutter &amp; Stroustrup, ROS, LINUX, Google's, Microsoft's, CERN's, GCC's, ARM's, LLVM's and probably thousands of others. Unfortunately most of these can't even agree on something as basic as how to name a class or a constant. This is probably due to the fact that these standards often carry lots of legacy issues due to supporting existing code bases. The intention behind this document is to create guidance that remains as close to ISO, Sutter &amp; Stroustrup and ROS while resolving as many conflicts, disadvantages and inconsistencies as possible among them.

Naming Conventions

Avoid using any sort of Hungarian notation on names and "_ptr" on pointers.

Code Element	Style	Comment
Namespace	under_scored	Differentiate from class names
Class name	CamelCase	To differentiate from STL types which ISO recommends (do not use "C" or "T" prefixes)
Function name	camelCase	Lower case start is almost universal except for .Net world
Parameters/Locals	under_scored	Vast majority of standards recommends this because _ is more readable to C++ crowd (although not much to Java/.Net crowd)
Member variables	**under_scored_with_**	The prefix _ is heavily discouraged as ISO has rules around reserving _identifiers, so we recommend suffix instead
Enums and its members	CamelCase	Most except very old standards agree with this one
Globals	g_under_scored	You shouldn't have these in first place!
Constants	UPPER_CASE	Very contentious and we just have to pick one here, unless if is a private constant in class or method, then use naming for Members or Locals
File names	Match case of class name in file	Lot of pro and cons either way but this removes inconsistency in auto generated code (important for ROS)

Header Files

Use a namespace qualified #ifdef to protect against multiple inclusion:

The reason we don't use #pragma once is because it's not supported if same header file exists at multiple places (which might be possible under ROS build system!).

Bracketing

Inside function or method body place curly bracket on same line. Outside that the Namespace, Class and methods levels use separate line. This is called K&amp;R style and its variants are widely used in C++ vs other styles which are more popular in other languages. Notice that curlies are not required if you have single statement, but complex statements are easier to keep correct with the braces.

Const and References

Religiously review all non-scalar parameters you declare to be candidate for const and references. If you are coming from languages such as C#/Java/Python, the most often mistake you would make is to pass parameters by value instead of `const T&;` Especially most of the strings, vectors and maps you want to pass as `const T&;` (if they are readonly) or `T&` (if they are writable). Also add `const` suffix to methods as much as possible.

Overriding

When overriding virtual method, use override suffix.

Pointers

This is really about memory management. A simulator has much performance critical code, so we try and avoid overloading the memory manager with lots of calls to new/delete. We also want to avoid too much copying of things on the stack, so we pass things by reference when ever possible. But when the object really needs to live longer than the call stack you often need to allocate that object on the heap, and so you have a pointer. Now, if management of the lifetime of that object is going to be tricky we recommend using C++ 11 smart pointers. But smart pointers do have a cost, so don't use them blindly everywhere. For private code where performance is paramount, raw pointers can be used. Raw pointers are also often needed when interfacing with legacy systems that only accept pointer types, for example, sockets API. But we try to wrap those legacy interfaces as much as possible and avoid that style of programming from leaking into the larger code base.

Religiously check if you can use const everywhere, for example, `const float * const xP`. Avoid using prefix or suffix to indicate pointer types in variable names, i.e. use `my_obj` instead of `myobj_ptr` except in cases where it might make sense to differentiate variables better, for example, `int mynum = 5; int* mynum_ptr = mynum;`

This is Too Short, ye?

Yes, and it's on purpose because no one likes to read 200 page coding guidelines. The goal here is to cover only most significant things which are already not covered by strict mode compilation in GCC and Level 4 warnings-as-errors in VC++. If you had like to know about how to write better code in C++, please see GotW and Effective Modern C++ book.

2.3.12 Contributing

How To

- Please read our short and sweet coding guidelines.
- For big changes such as adding new feature or refactoring, file an issue first. We should talk!
- Use our recommended development workflow to make changes and test it.
- Use usual steps to make contributions just like other GitHub projects. If you are not familiar with Git Branch-Rebase-Merge workflow, please read this first.

Do and Don't

- Rebase your branch frequently with master (once every 2-3 days is ideal).
- Use same style and formatting as rest of code even if it's not your preferred one.
- Change any documentation that goes with code changes.
- Do not include OS specific header files.
- Keep your pull request small, ideally under 10 files.

2.3.13 How to Create Issue or Ask Question Effectively

AirSim is open source project and contributors like you keeps it going. It is important to respect contributors time and effort when you are asking a question or filing an issue. Your chances of receiving helpful response would increase if you follow below guidelines:

DOs

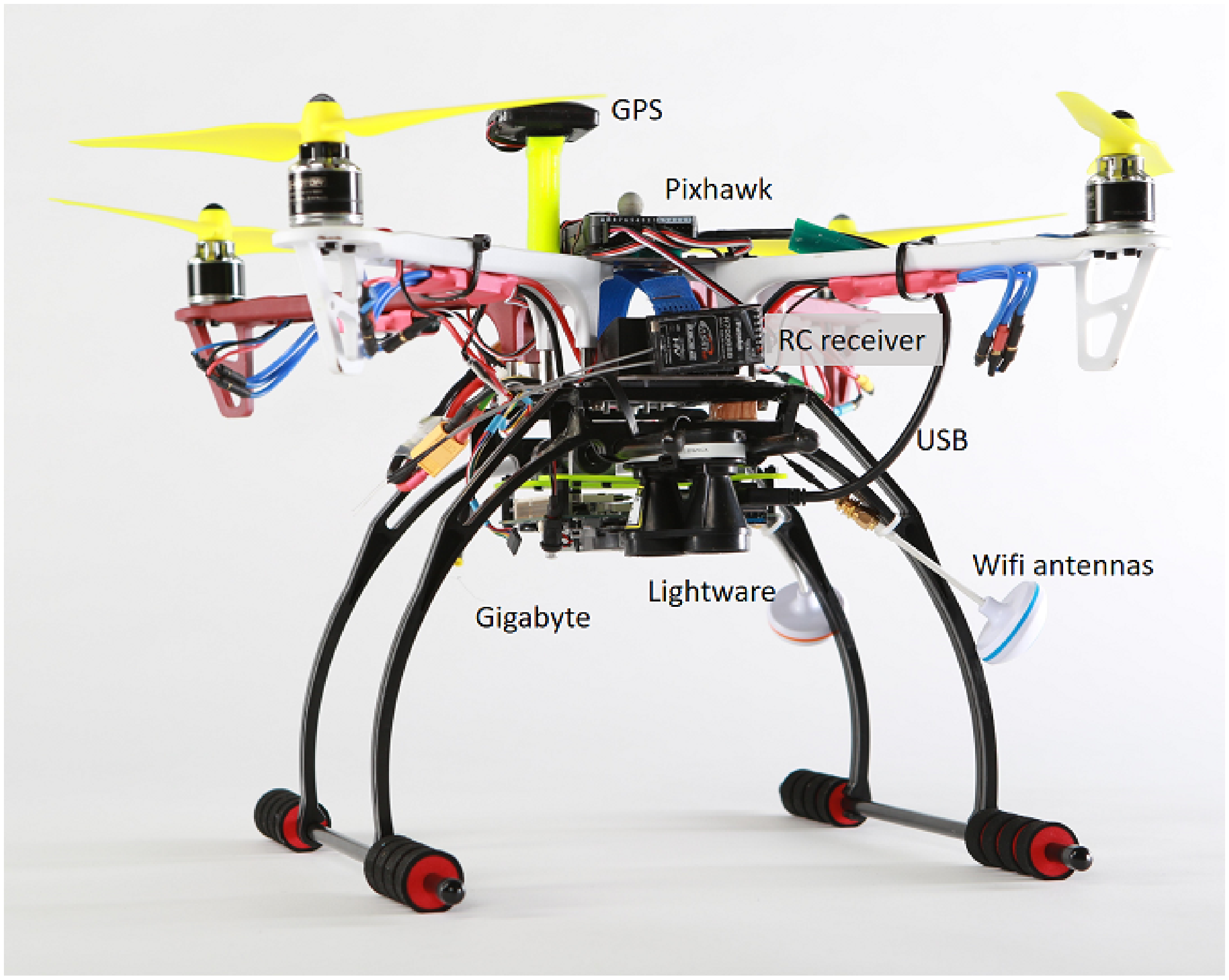
- Search issues to see if someone already has asked it.
- Chose title that is short and summarizes well.
- Copy and paste full error message.
- Precisely describe steps you used that produced the error message or symptom.
- Describe what vehicle, mode, OS, AirSim version and other settings you are using.
- Copy and paste minimal version of code that reproduces the problem.
- Tell us what the goal you want to achieve or expected output.
- Tell us what you did so far to debug this issue.

DONT'S

- Do not use "Please help" etc in the title. See above.
- Do not copy and paste screen shot of error message. Copy and paste text.
- Do not use "it doesn't work". Precisely state what is the error message or symptom.
- Do not ask to write code for you. Contribute!

2.3.14 AirLib on a Real Drone

The AirLib library can be compiled and deployed on the companion computer on a real drone. For our testing, we mounted a Gigabyte Brix BXi7-5500 ultra compact PC on the drone connected to the Pixhawk flight controller over USB. The Gigabyte PC is running Ubuntu, so we are able to SSH into it over Wi-Fi:



Once connected you can run MavLinkTest with this command line:

And this will produce a log file of the flight which can then be used for playback in the simulator.

You can also add `-proxy:192.168.1.100:14550` to connect MavLinkTest to a remote computer where you can run QGroundControl or our PX4 Log Viewer which is another handy way to see what is going on with your drone.

MavLinkTest then has some simple commands for testing your drone, here's a simple example of some commands:

This will arm the drone, takeoff of 5 meters, then do an orbit pattern radius 10 meters, at 2 m/s. Type '?' to find all available commands.

Note: Some commands (for example, `orbit`) are named differently and have different syntax in MavLinkTest and DroneShell (for example, `circlebypath -radius 10 -velocity 21`).

When you land the drone you can stop MavLinkTest and copy the *.mavlink log file that was generated.

2.3.15 DroneServer and DroneShell

Once you are happy that the MavLinkTest is working, you can also run DroneServer and DroneShell as follows. First, run MavLinkTest with a local proxy to send everything to DroneServer:

Change ~/Documents/AirSim/settings.json to say "serial":false, because we want DroneServer to look for this UDP connection.

Lastly, you can now connect DroneShell to this instance of DroneServer and use the DroneShell commands to fly your drone:

PX4 Specific Tools

You can run the MavlinkCom library and MavLinkTest app to test the connection between your companion computer and flight controller.

How Does This Work?

AirSim uses MavLinkCom component developed by @lovettchris. The MavLinkCom has a proxy architecture where you can open a connection to PX4 either using serial or UDP and then other components share this connection. When PX4 sends MavLink message, all components receive that message. If any component sends a message then it's received by PX4 only. This allows you to connect any number of components to PX4 This code opens a connection for LogViewer and QGC. You can add something more if you like.

If you want to use QGC + AirSim together than you will need QGC to let own the serial port. QGC opens up TCP connection that acts as a proxy so any other component can connect to QGC and send MavLinkMessage to QGC and then QGC forwards that message to PX4. So you tell AirSim to connect to QGC and let QGC own serial port.

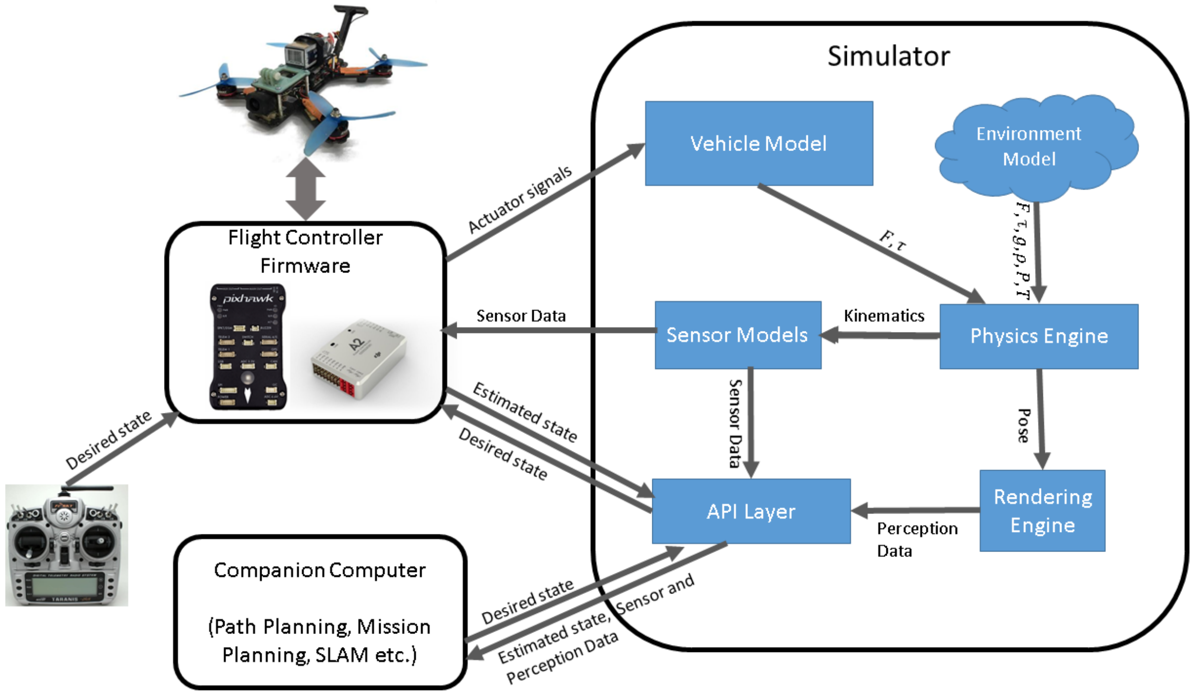
For companion board, the way we did it earlier was to have Gigabyte Brix on the drone. This x86 full-fledged computer that will connect to PX4 through USB. We had Ubuntu on Brix and ran DroneServer. The DroneServer created an API endpoint that we can talk to via C++ client code (or Python code) and it translated API calls to MavLink messages. That way you can write your code against the same API, test it in the simulator and then run the same code on an actual vehicle. So the companion computer has DroneServer running along with client code.

2.3.16 Paper

You can read more about our architecture and design in our paper (work in progress). You may cite this as,

2.3.17 Architecture

Below is high level overview of how different components interact with each other.



2.3.18 Development Workflow

Below is the guide on how to perform different development activities while working with AirSim. If you are new to Unreal Engine based projects and want to contribute to AirSim or make your own forks for your custom requirements, this might save you some time.

Development Environment

OS

We highly recommend Windows 10 and Visual Studio 2017 as your development environment. The support for other OSes and IDE is unfortunately not as mature on the Unreal Engine side and you may risk severe loss of productivity trying to do workarounds and jumping through the hoops.

Hardware

We recommend GPUs such as NVidia 1080 or NVidia Titan series with powerful desktop such as one with 64GB RAM, 6+ cores, SSDs and 2-3 displays (ideally 4K). We have found HP Z840 work quite well for our needs. The development experience on high-end laptops is generally sub-par compared to powerful desktops however they might be useful in a pinch. You generally want laptops with discrete NVidia GPU (at least M2000 or better) with 64GB RAM, SSDs and hopefully 4K display. We have found models such as Lenovo P50 work well for our needs. Laptops with only integrated graphics might not work well.

Updating and Changing AirSim Code

Overview

AirSim is designed as plugin. This means it can't run by itself, you need to put it in an Unreal project (we call it "environment"). So building and testing AirSim has two steps: (1) build the plugin (2) deploy plugin in Unreal project and run the project.

The first step is accomplished by build.cmd available in AirSim root. This command will update everything you need for the plugin in the `Unreal\Plugins` folder. So to deploy the plugin, you just need to copy `Unreal\Plugins` folder in to your Unreal project folder. Next you should remove all intermediate files in your Unreal project and then regenerate .sln file for your Unreal project. To do this, we have two handy .bat files in `Unreal\Environments\Blocks` folder: `clean.bat` and `GenerateProjectFiles.bat`. So just run these bat files in sequence from root of your Unreal project. Now you are ready to open new .sln in Visual Studio and press F5 to run it.

Steps

Below are the steps we use to make changes in AirSim and test them out. The best way to do development in AirSim code is to use Blocks project. This is the light weight project so compile time is relatively faster. Generally the workflow is,

Above commands first builds the AirSim plugin and then deploys it to Blocks project using handy `update_from_git.bat`. Now you can work inside Visual Studio solution, make changes to the code and just run F5 to build, run and test your changes. The debugging, break points etc should work as usual.

After you are done with you code changes, you might want to push your changes back to AirSim repo or your own fork or you may deploy the new plugin to your custom Unreal project. To do this, go back to command prompt and first update the AirSim repo folder:

Above command will transfer your code changes from Unreal project folder back to `Unreal\Plugins` folder. Now your changes are ready to be pushed to AirSim repo or your own fork. You can also copy `Unreal\Plugins` to your custom Unreal engine project and see if everything works in your custom project as well.

Take Away

> Once you understand how Unreal Build system and plugin model works as well as why we are doing above steps, you should feel quite comfortable in following this workflow. Don't be afraid of opening up .bat files to peek inside and see what its doing. They are quite minimal and straightforward (except, of course, build.cmd - don't look in to that one).

FAQ

When you press F5 or F6 in Visual Studio to start build, the Unreal Build system kicks in and it tries to find out if any files are dirty and what it needs to build. Unfortunately, it often fails to recognize dirty files that is not the code that uses Unreal headers and object hierarchy. So, the trick is to just make some file dirty that Unreal Build system always recognizes. My favorite one is AirSimGameMode.cpp. Just insert a line, delete it and save the file.

Don't do that! Unreal Build system *assumes* that you are using Visual Studio and it does bunch of things to integrate with Visual Studio. If you do insist on using other editors then look up how to do command line builds in Unreal projects OR see docs on your editor on how it can integrate with Unreal build system OR run `clean.bat` + `GenerateProjectFiles.bat` to make sure VS solution is in sync.

It won't! While you are indeed using Visual Studio solution, remember that this solution was actually generated by Unreal Build system. If you want to add new files in your project, first shut down Visual Studio, add an empty file at desired location and then run `GenerateProjectFiles.bat` which will scan all files in your project and then re-create the .sln file. Now open this new .sln file and you are in business.

First make sure your project's .uproject file is referencing the plugin. Then make sure you have run `clean.bat` and then `GenerateProjectFiles.bat` as described in Overview above.

You are in luck! We have `build_all_ue_projects.bat` which exactly does that. Don't treat it as black box (at least not yet), open it up and see what it does. It has 4 variables that are being set from command line args. If these args is not supplied they are set to default values in next set of statements. You might want to change default values for the paths. This batch file builds AirSim plugin, deploys it to all listed projects (see CALL statements later in the batch file), runs packaging for those projects and puts final binaries in specified folder - all in one step! This is what we use to create our own binary releases.

Before making any changes make sure you have created your feature branch. After you test your code changes in Blocks environment, follow the usual steps to make contributions just like any other GitHub projects. If you are not familiar with Git Branch-Rebase-Merge workflow, please read this first.

2.3.19 FAQ

General

Unreal editor is slow when it is not the active window

Go to Edit/Editor Preferences, select "All Settings" and type "CPU" in the search box. It should find the setting titled "Use Less CPU when in Background", and you want to uncheck this checkbox.

My mouse disappears in Unreal

Yes, Unreal steals the mouse, and we don't draw one. So to get your mouse back just use Alt+TAB to switch to a different window. To avoid this entirely, go to Project settings in Unreal Editor, go to Input tab and disable all settings for mouse capture.

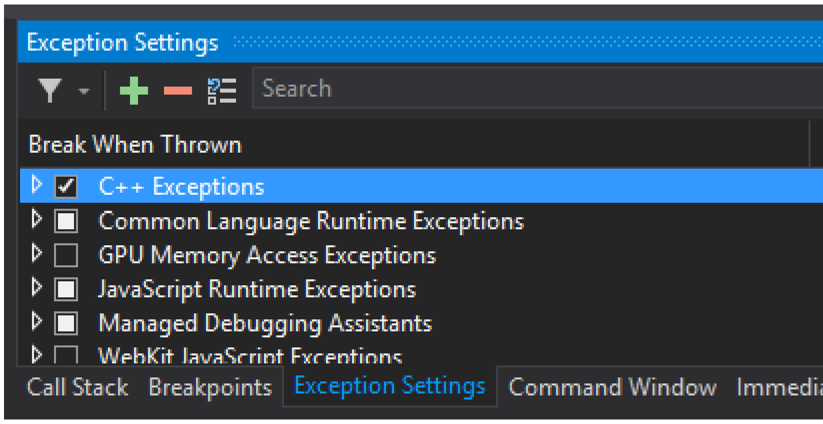
Where is the setting file and how do I modify it?

AirSim will create empty settings file at `~/Documents/AirSim/settings.json`. You can view the available settings options.

How do I arm my drone?

If you're using simple_flight, your vehicle is already armed and ready to fly. For PX4 you can arm by holding both sticks on remote control down and to the center.

Something went wrong. How do I debug?

First turn on C++ exceptions from the Exceptions window: 

Exception Settings

Search

Break When Thrown

- ☑ C++ Exceptions
- ■ Common Language Runtime Exceptions
- ☐ GPU Memory Access Exceptions
- ■ JavaScript Runtime Exceptions
- ■ Managed Debugging Assistants
- ☐ WebKit JavaScript Exceptions

and copy the stack trace of all exceptions you see there during execution that look relevant (for example, there might be an initial exception from VSPerf140 that you can ignore) then paste these call stacks into a new AirSim GitHub issue, thanks.

What do the colors mean in the Segmentation View ?

See Camera Views for information on the camera views and how to change them.

Unreal 4.xx doesn't look as good as 4.yy

Unreal 4.15 added the ability for Foliage LOD dithering to be disabled on a case-by-case basis by unchecking the `Dithered LOD Transition` checkbox in the foliage materials. Note that all materials used on all LODs need to have the checkbox checked in order for dithered LOD transitions to work. When checked the transition of generated foliage will be a lot smoother and will look better than 4.14.

Can I use an XBox controller to fly?

See XBox controller for details.

Can I build a hexacopter with AirSim?

See how to build a hexacopter.

How do I use AirSim with multiple vehicles?

Here is multi-vehicle setup guide.

What computer do you need?

It depends on how big your Unreal Environment is. The Blocks environment that comes with AirSim is very basic and works on typical laptops. The Modular Neighborhood Pack that we use ourselves for research requires GPUs with at least 4GB of RAM. The Open World environment needs GPU with 8GB RAM. Our typical development machines have 32GB of RAM and NVIDIA TitanX and a fast hard drive.

How do I report issues?

It's a good idea to include your configuration like below. If you can also include logs, that could also expedite the investigation.

If you have modified the default `~/Document/AirSim/settings.json`, please include your settings also.

If you are using PX4 then try to capture log from MavLink or PX4.

File an issue through GitHub Issues.

Others

Linux Build FAQ Windows Build FAQ PX4 Setup FAQ Remote Control FAQ Unreal Blocks Environment FAQ Unreal Custom Environment FAQ

2.3.20 Flight Controller

What is Flight Controller?

"Wait!" you ask, "Why do you need flight controller for a simulator?". The primary job of flight controller is to take in *desired state* as input, estimate *actual state* using sensors data and then drive the actuators in such a way so that actual state comes as close to the desired state. For quadrotors, desired state can be specified as roll, pitch and yaw, for example. It then estimates actual roll, pitch and yaw using gyroscope and accelerometer. Then it generates appropriate motor signals so actual state becomes desired state. You can find more in-depth in our paper.

How Simulator uses Flight Controller?

Simulator consumes the motor signals generated by flight controller to figure out force and thrust generated by each actuator (i.e. propellers in case of quadrotor). This is then used by the physics engine to compute the kinetic properties of the vehicle. This in turn generates simulated sensor data and feed it back to the flight controller. You can find more in-depth in our paper.

What is Hardware- and Software-in-Loop?

Hardware-in-Loop (HITL or HIL) means flight controller runs in actual hardware such as Naze32 or Pixhawk chip. You then connect this hardware to PC using USB port. Simulator talks to the device to retrieve actuator signals and send it simulated sensor data. This is obviously as close as you can get to real thing. However, it typically requires more steps to set up and usually hard to debug. One big issue is that simulator clock and device clock runs on their own speed and accuracy. Also, USB connection (which is usually only USB 2.0) may not be enough for real-time communication.

In "software-in-loop" simulation (SITL or SIL) mode the firmware runs in your computer as opposed to separate board. This is generally fine except that now you are not touching any code paths that are specific to your device. Also, none of your code now runs with real-time clock usually provided by specialized hardware board. For well-designed flight controllers with software clock, these are usually not concerning issues.

What Flight Controllers are Supported?

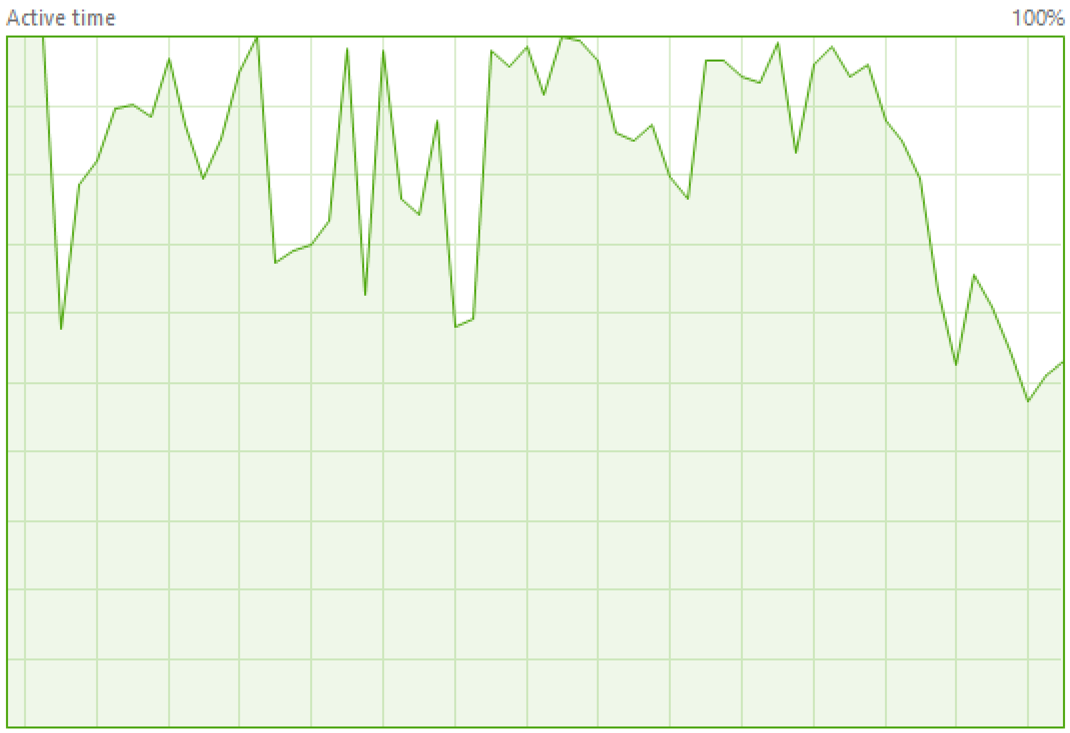
AirSim has built-in flight controller called simple_flight and it is used by default. You don't need to do anything to use or configure it. AirSim also supports PX4 as another flight controller for advanced users. In the future, we also plan to support ROSFlight and Hackflight.

Using AirSim Without Flight Controller

Yes, now it's possible to use AirSim without flight controller. Please see the instructions here for how to use so-called "Computer Vision" mode. If you don't need vehicle dynamics, we highly recommend using this mode.

2.3.21 Busy Hard Drive

It is not required, but we recommend running your Unreal Environment on a Solid State Drive (SSD). Between debugging, logging, and Unreal asset loading the hard drive can become your bottle neck. It is normal that your hard drive will be slammed while Unreal is loading the environment, but if your hard drive performance looks like this while the Unreal game is running then you will probably not get a good flying experience.

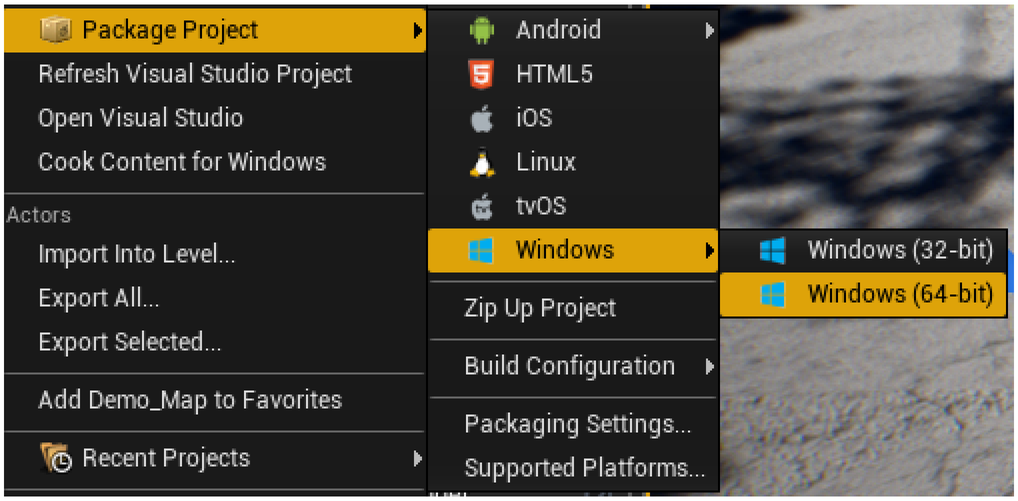


In fact, if the hard drive is this busy, chances are the drone will not fly properly at all. For some unknown reason this I/O bottle neck also interferes with the drone control loop and if that loop doesn't run at a high rate (300-500 Hz) then the drone will not fly. Not surprising, the control loop inside the PX4 firmware that runs on a Pixhawk flight controller runs at 1000 Hz.

Reducing I/O

If you can't whip off to Fry's Electronics and pick up an overpriced super fast SSD this weekend, then the following steps can be taken to reduce the hard drive I/O:

1. First run the Unreal Environment using Cooked content outside of the UE Editor or any debugging environment, and package the content to your fastest SSD drive. You can do that using this menu option:



1. If you must use the UE editor (because you are actively modifying game assets), then at least don't run that in a debugger. If you are using Visual Studio use start without debugging.
2. If you must debug the app, and you are using Visual Studio debugger, stop then Visual Studio from logging Intellitrace information. Go to Tools/Options/Debugging/Intellitrace, and turn off the main checkbox.
3. Turn off any Unreal Analytics that your environment may have enabled, especially any file logging.

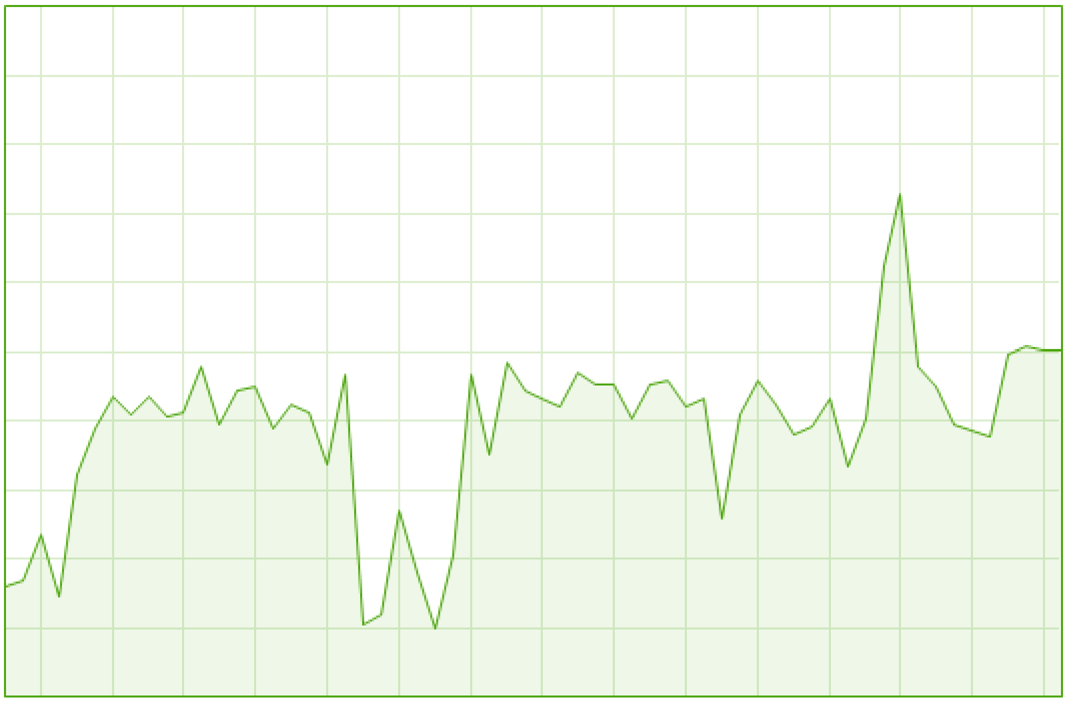
I/O from Page Faults

If your system is running out of RAM it may start paging memory to disk. If your operating system has enabled paging to disk, make sure it is paging to your fastest SSD. Or if you have enough RAM disable paging all together. In fact, if you disable paging and the game stops working you will know for sure you are running out of RAM.

Obviously, shutting down any other unnecessary apps should also free up memory so you don't run out.

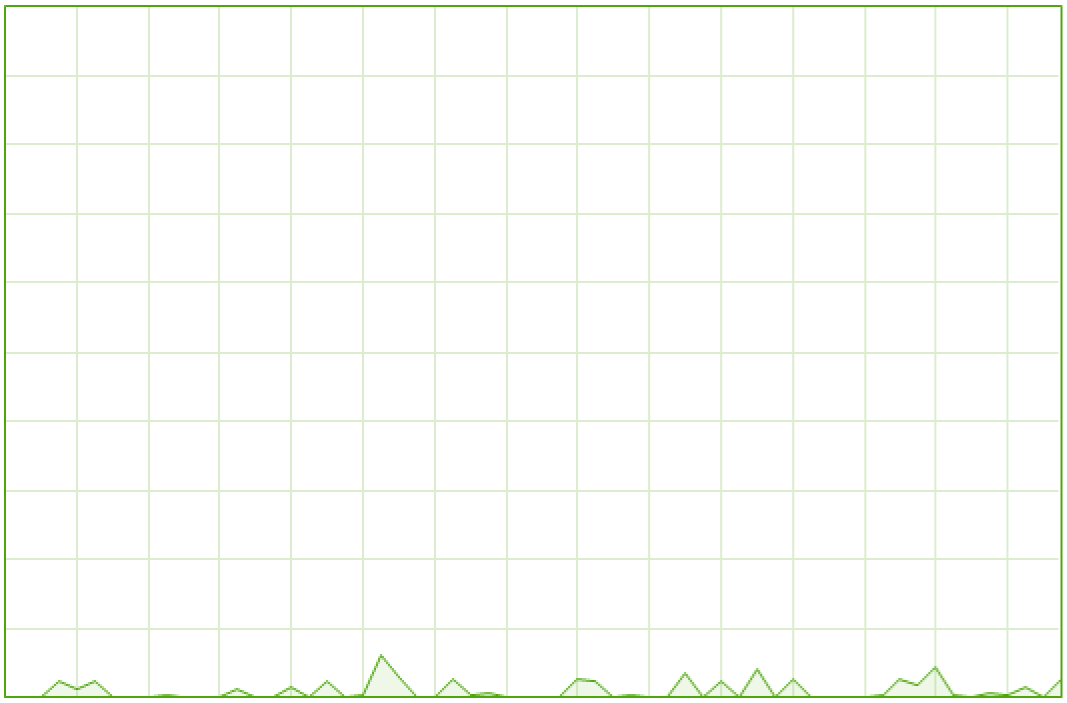
Ideal Runtime performance

This is what my slow hard drive looks like when flying from UE editor. You can see it's very busy, but the drone still



flies ok:

This is what my fast SSD looks like when the drone is flying in an Unreal Cooked app (no UE editor, no debugger). Not surprisingly it is flying perfectly in this case:



2.3.22 Hello Drone

How does Hello Drone work?

Hello Drone uses the RPC client to connect to the RPC server that is automatically started by the AirSim. The RPC server routes all the commands to a class that implements DroneControlBase. In essence, DroneControlBase defines our abstract interface for getting data from the quadrotor and sending back commands. We currently have concrete implementation for DroneControlBase for MavLink based vehicles. The implementation for DJI drone platforms, specifically Matrice, is in works.

2.3.23 Image APIs

Please read general API doc first if you are not familiar with AirSim APIs.

Getting a Single Image

Here's a sample code to get a single image from camera named "0". The returned value is bytes of png format image. To get uncompressed and other format as well as available cameras please see next sections.

Python

```python
import airsim #pip install airsim

# for car use CarClient()
client = airsim.MultirotorClient()

png_image = client.simGetImage("0", airsim.ImageType.Scene)
# do something with image
```

C++

```cpp
#include "vehicles/multirotor/api/MultirotorRpcLibClient.hpp"

int getOneImage()
{
    using namespace std;
    using namespace msr::airlib;

    //for car use CarRpcLibClient
    msr::airlib::MultirotorRpcLibClient client;

    vector<uint8_t> png_image = client.simGetImage("0",␣
↪VehicleCameraBase::ImageType::Scene);
    //do something with images
}
```

Getting Images with More Flexibility

The `simGetImages` API which is slightly more complex to use than `simGetImage` API, for example, you can get left camera view, right camera view and depth image from left camera in a single API call. The `simGetImages`

API also allows you to get uncompressed images as well as floating point single channel images (instead of 3 channel (RGB), each 8 bit).

Python

```
import airsim #pip install airsim

# for car use CarClient()
client = airsim.MultirotorClient()

responses = client.simGetImages([
    # png format
    airsim.ImageRequest(0, airsim.ImageType.Scene),
    # uncompressed RGBA array bytes
    airsim.ImageRequest(1, airsim.ImageType.Scene, False, False),
    # floating point uncompressed image
    airsim.ImageRequest(1, airsim.ImageType.DepthPlanner, True)])

 # do something with response which contains image data, pose, timestamp etc
```

Using AirSim Images with NumPy

If you plan to use numpy for image manipulation, you should get uncompressed RGBA image and then convert to numpy like this:

```
responses = client.simGetImages([ImageRequest("0", airsim.ImageType.Scene, False,␣
↪False)])
response = responses[0]

# get numpy array
img1d = np.fromstring(response.image_data_uint8, dtype=np.uint8)

# reshape array to 4 channel image array H X W X 4
img_rgba = img1d.reshape(response.height, response.width, 4)

# original image is fliped vertically
img_rgba = np.flipud(img_rgba)

# just for fun add little bit of green in all pixels
img_rgba[:,:,1:2] = 100

# write to png
airsim.write_png(os.path.normpath(filename + '.greener.png'), img_rgba)
```

Quick Tips

- The API `simGetImage` returns `binary string literal` which means you can simply dump it in binary file to create a .png file. However if you want to process it in any other way than you can handy function `airsim.string_to_uint8_array`. This converts binary string literal to NumPy uint8 array.
- The API `simGetImages` can accept request for multiple image types from any cameras in single call. You can specify if image is png compressed, RGB uncompressed or float array. For png compressed images, you get

`binary string literal`. For float array you get Python list of float64. You can convert this float array to NumPy 2D array using

C++

```cpp
int getStereoAndDepthImages()
{
    using namespace std;
    using namespace msr::airlib;

    typedef VehicleCameraBase::ImageRequest ImageRequest;
    typedef VehicleCameraBase::ImageResponse ImageResponse;
    typedef VehicleCameraBase::ImageType ImageType;

    //for car use
    //msr::airlib::CarRpcLibClient client;
    msr::airlib::MultirotorRpcLibClient client;

    //get right, left and depth images. First two as png, second as float16.
    vector<ImageRequest> request = {
        //png format
        ImageRequest("0", ImageType::Scene),
        //uncompressed RGBA array bytes
        ImageRequest("1", ImageType::Scene, false, false),
        //floating point uncompressed image
        ImageRequest("1", ImageType::DepthPlanner, true)
    };

    const vector<ImageResponse>& response = client.simGetImages(request);
    //do something with response which contains image data, pose, timestamp etc
}
```

Ready to Run Complete Examples

Python

For a more complete ready to run sample code please see sample code in AirSimClient project for multirotors or HelloCar sample. This code also demonstrates simple activities such as saving images in files or using `numpy` to manipulate images.

C++

For a more complete ready to run sample code please see sample code in HelloDrone project for multirotors or HelloCar project.

See also other example code that generates specified number of stereo images along with ground truth depth and disparity and saving it to pfm format.

Available Cameras

Car

The cameras on car can be accessed by following names in API calls: `front_center`, `front_right`, `front_left`, `fpv` and `back_center`. Here FPV camera is driver's head position in the car.

Multirotor

The cameras in CV mode can be accessed by following names in API calls: `front_center`, `front_right`, `front_left`, `bottom_center` and `back_center`.

Computer Vision Mode

Camera names are same as in multirotor.

Backward compatibility for camera names

Before AirSim v1.2, cameras were accessed using ID numbers instead of names. For backward compatibility you can still use following ID numbers for above camera names in same order as above: `"0"`, `"1"`, `"2"`, `"3"`, `"4"`. In addition, camera name `""` is also available to access the default camera which is generally the camera `"0"`.

"Computer Vision" Mode

You can use AirSim in so-called "Computer Vision" mode. In this mode, physics engine is disabled and there is no vehicle, just cameras. You can move around using keyboard (use F1 to see help on keys). You can press Record button to continuously generate images. Or you can call APIs to move cameras around and take images.

To active this mode, edit settings.json that you can find in your `Documents\AirSim` folder (or `~/Documents/AirSim` on Linux) and make sure following values exist at root level:

```
{
  "SettingsVersion": 1.2,
  "SimMode": "ComputerVision"
}
```

Here's the Python code example to move camera around and capture images.

This mode was inspired from UnrealCV project.

Setting Pose in Computer Vision Mode

To move around the environment using APIs you can use `simSetVehiclePose` API. This API takes position and orientation and sets that on the invisible vehicle where the front-center camera is located. All rest of the cameras move along keeping the relative position. If you don't want to change position (or orientation) then just set components of position (or orientation) to floating point nan values. The `simGetVehiclePose` allows to retrieve the current pose. You can also use `simGetGroundTruthKinematics` to get the quantities kinematics quantities for the movement. Many other non-vehicle specific APIs are also available such as segmentation APIs, collision APIs and camera APIs.

Camera APIs

The `simGetCameraInfo` returns the pose (in world frame, NED coordinates, SI units) and FOV (in degrees) for the specified camera. Please see example usage.

The `simSetCameraOrientation` sets the orientation for the specified camera as quaternion in NED frame. The handy `airsim.to_quaternion()` function allows to convert pitch, roll, yaw to quaternion. For example, to set camera-0 to 15-degree pitch, you can use:

Gimbal

You can set stabilization for pitch, roll or yaw for any camera using settings.

Please see example usage.

Changing Resolution and Camera Parameters

To change resolution, FOV etc, you can use settings.json. For example, below addition in settings.json sets parameters for scene capture and uses "Computer Vision" mode described above. If you omit any setting then below default values will be used. For more information see settings doc. If you are using stereo camera, currently the distance between left and right is fixed at 25 cm.

```
{
  "SettingsVersion": 1.2,
  "CameraDefaults": {
      "CaptureSettings": [
        {
          "ImageType": 0,
          "Width": 256,
          "Height": 144,
          "FOV_Degrees": 90,
          "AutoExposureSpeed": 100,
          "MotionBlurAmount": 0
        }
    ]
  },
  "SimMode": "ComputerVision"
}
```

What Does Pixel Values Mean in Different Image Types?

Available ImageType Values

```
Scene = 0,
DepthPlanner = 1,
DepthPerspective = 2,
DepthVis = 3,
DisparityNormalized = 4,
Segmentation = 5,
SurfaceNormals = 6,
Infrared = 7
```

DepthPlanner and DepthPerspective

You normally want to retrieve the depth image as float (i.e. set `pixels_as_float = true`) and specify `ImageType = DepthPlanner` or `ImageType = DepthPerspective` in `ImageRequest`. For `ImageType = DepthPlanner`, you get depth in camera plan, i.e., all points that are in plan parallel to camera have same depth. For `ImageType = DepthPerspective`, you get depth from camera using a projection ray that hits that pixel. Depending on your use case, planner depth or perspective depth may be the ground truth image that you want. For example, you may be able to feed perspective depth to ROS package such as `depth_image_proc` to generate a point cloud. Or planner depth may be more compatible with estimated depth image generated by stereo algorithms such as SGM.

DepthVis

When you specify `ImageType = DepthVis` in `ImageRequest`, you get an image that helps depth visualization. In this case, each pixel value is interpolated from black to white depending on depth in camera plane in meters. The pixels with pure white means depth of 100m or more while pure black means depth of 0 meters.

DisparityNormalized

You normally want to retrieve disparity image as float (i.e. set `pixels_as_float = true` and specify `ImageType = DisparityNormalized` in `ImageRequest`) in which case each pixel is `(Xl - Xr)/Xmax`, which is thereby normalized to values between 0 to 1.

Segmentation

When you specify `ImageType = Segmentation` in `ImageRequest`, you get an image that gives you ground truth segmentation of the scene. At the startup, AirSim assigns value 0 to 255 to each mesh available in environment. This value is than mapped to a specific color in the pallet. The RGB values for each object ID can be found in this file.

You can assign a specific value (limited to the range 0-255) to a specific mesh using APIs. For example, below Python code sets the object ID for the mesh called "Ground" to 20 in Blocks environment and hence changes its color in Segmentation view:

```
success = client.simSetSegmentationObjectID("Ground", 20);
```

The return value is a boolean type that lets you know if the mesh was found.

Notice that typical Unreal environments, like Blocks, usually have many other meshes that comprises of same object, for example, "Ground_2", "Ground_3" and so on. As it is tedious to set object ID for all of these meshes, AirSim also supports regular expressions. For example, the code below sets all meshes which have names starting with "ground" (ignoring case) to 21 with just one line:

```
success = client.simSetSegmentationObjectID("ground[\w]*", 21, True);
```

The return value is true if at least one mesh was found using regular expression matching.

It is recommended that you request uncompressed image using this API to ensure you get precise RGB values for segmentation image:

```
responses = client.simGetImages([ImageRequest(0, AirSimImageType.Segmentation, False,␣
↪False)])
img1d = np.fromstring(response.image_data_uint8, dtype=np.uint8) #get numpy array
```

(continues on next page)

(continued from previous page)

```
img_rgba = img1d.reshape(response.height, response.width, 4) #reshape array to 4␣
↪channel image array H X W X 4
img_rgba = np.flipud(img_rgba) #original image is fliped vertically

#find unique colors
print(np.unique(img_rgba[:,:,0], return_counts=True)) #red
print(np.unique(img_rgba[:,:,1], return_counts=True)) #green
print(np.unique(img_rgba[:,:,2], return_counts=True)) #blue
```

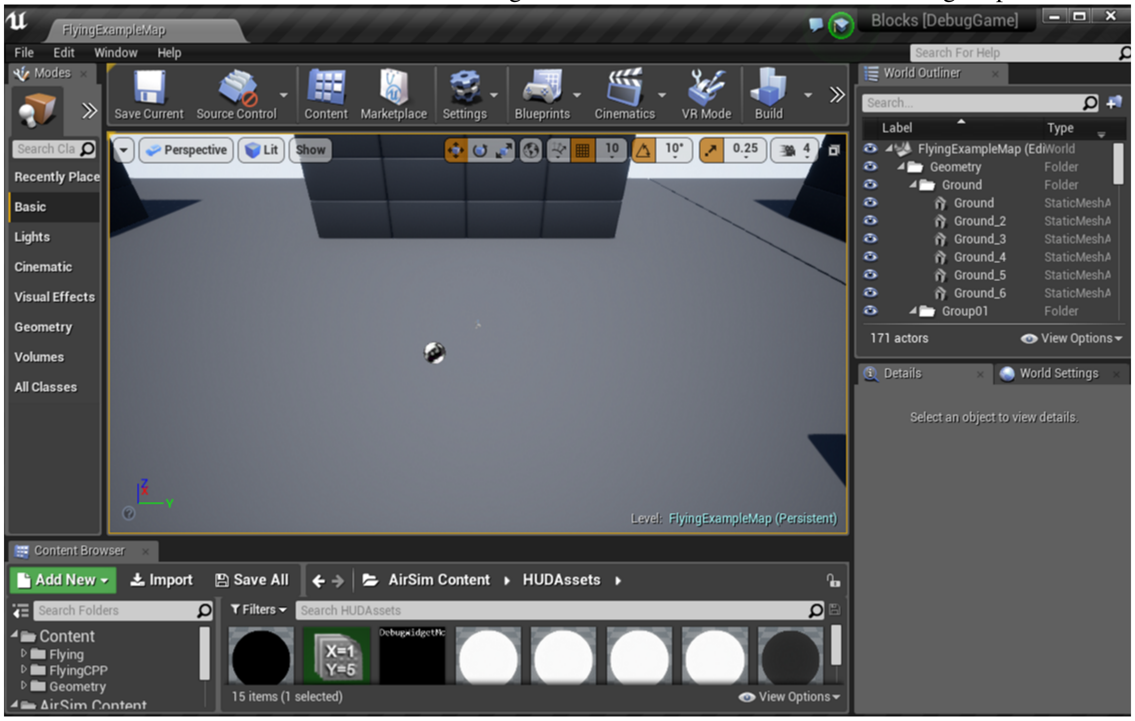
A complete ready-to-run example can be found in segmentation.py.

Unsetting object ID

An object's ID can be set to -1 to make it not show up on the segmentation image.

How to Find Mesh Names?

To get desired ground truth segmentation you will need to know the names of the meshes in your Unreal environment. To do this, you will need to open up Unreal Environment in Unreal Editor and then inspect the names of the meshes you are interested in using the World Outliner. For example, below we see the mesh names for he ground in Blocks environment in right panel in the editor:



Label	Type
FlyingExampleMap (Edi	World
Geometry	Folder
Ground	Folder
Ground	StaticMeshA
Ground_2	StaticMeshA
Ground_3	StaticMeshA
Ground_4	StaticMeshA
Ground_5	StaticMeshA
Ground_6	StaticMeshA
Group01	Folder

171 actors

If you don't know how to open Unreal Environment in Unreal Editor then try following the guide for building from source.

Once you decide on the meshes you are interested, note down their names and use above API to set their object IDs. There are few settings available to change object ID generation behavior.

Changing Colors for Object IDs

At present the color for each object ID is fixed as in this palate. We will be adding ability to change colors for object IDs to desired values shortly. In the meantime you can open the segmentation image in your favorite image editor and get the RGB values you are interested in.

Startup Object IDs

At the start, AirSim assigns object ID to each object found in environment of type `UStaticMeshComponent` or `ALandscapeProxy`. It then either uses mesh name or owner name (depending on settings), lower cases it, removes any chars below ASCII 97 to remove numbers and some punctuations, sums int value of all chars and modulo 255 to generate the object ID. In other words, all object with same alphabet chars would get same object ID. This heuristic is simple and effective for many Unreal environments but may not be what you want. In that case, please use above APIs to change object IDs to your desired values. There are few settings available to change this behavior.

Getting Object ID for Mesh

The `simGetSegmentationObjectID` API allows you get object ID for given mesh name.

Infrared

Currently this is just a map from object ID to grey scale 0-255. So any mesh with object ID 42 shows up with color (42, 42, 42). Please see *segmentation section* for more details on how to set object IDs. Typically noise setting can be applied for this image type to get slightly more realistic effect. We are still working on adding other infrared artifacts and any contributions are welcome.

Example Code

A complete example of setting vehicle positions at random locations and orientations and then taking images can be found in GenerateImageGenerator.hpp. This example generates specified number of stereo images and ground truth disparity image and saving it to pfm format.

2.3.24 How to Use Lidar in AirSim

AirSim supports Lidar for multirotors and cars.

The enablement of lidar and the other lidar settings can be configured via AirSimSettings json. Please see general sensors for information on configruation of general/shared sensor settings.

Enabling lidar on a vehicle

- By default, lidars are not enabled. To enable lidar, set the SensorType and Enabled attributes in settings json. .. code-block:

```
"Lidar1": {
     "SensorType": 6,
     "Enabled" : true,
```

- Multiple lidars can be enabled on a vehicle.

Lidar configuration

The following parameters can be configured right now via settings json.

Parameter	Description
NumberOfChannels	Number of channels/lasers of the lidar
Range	Range, in meters
PointsPerSecond	Number of points captured per second
RotationsPerSecond	Rotations per second
VerticalFOVUpper	Vertical FOV upper limit for the lidar, in degrees
VerticalFOVLower	Vertical FOV lower limit for the lidar, in degrees
X Y Z	Position of the lidar relative to the vehicle (in NED, in meters)
Roll Pitch Yaw	Roation of the lidar relative to the vehicle (in degrees)

e.g.,

Server side visualization for debugging

Be default, the lidar points are not drawn on the viewport. To enable the drawing of hit laser points on the viewport, please enable setting 'DrawDebugPoints' via settings json. e.g.,

Client API

Use `getLidarData()` API to retrieve the Lidar data.

- The API returns a Point-Cloud as a flat array of floats along with a timestamp of the capture.
- The floats represent [x,y,z] coordinate for each point hit within the range in the last scan.
- The coordinates are in the local vehicle NED like all other AirSim APIs.

Python Examples

drone_lidar.py car_lidar.py

Coming soon

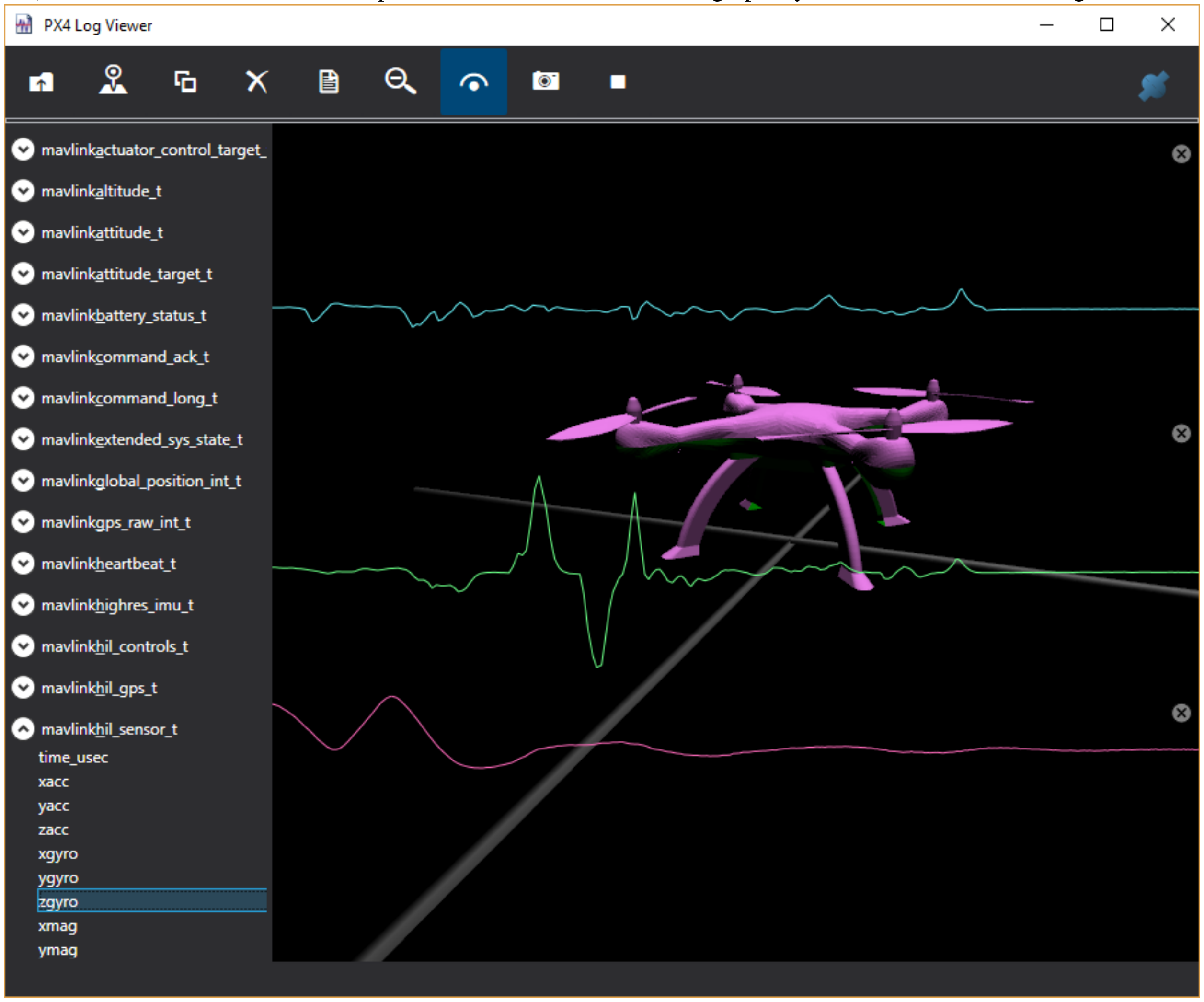
- Visualization of lidar data on client side.

2.3.25 Log Viewer

The LogViewer is a Windows WPF app that presents the MavLink streams that it is getting from the Unreal Simulator. You can use this to monitor what is happening on the drone while it is flying. For example, the picture below shows a real time graph of the x, y an z gyro sensor information being generated by the simulator.

Usage

To use this LogViewer, connect the simulator `before` you run the simulation. Simply press the blue connector button on the top right corner of the window, select the Socket `tab`, enter the port number 14388, and your `localhost` network. Then press the record button (triangle on the right hand side of the toolbar). Now start the simulator, pick some mavlink items to graph, you should see something like this:



The drone view here is the actual estimated position coming from the PX4, so that is a great way to check whether the PX4 is in sync with the simulator. Sometimes you can see some drift here as the attitude estimation catches up with reality, this is more visible after a bad crash.

Installation

If you can't build the LogViewer.sln, there is also a click once installer.

Configuration

The magic port number 14388 can be configured in the simulator by editing the settings.json file.

2.3.26 Multiple Vehicles in AirSim

Since release 1.2, AirSim is fully enabled for multiple vehicles. This capability allows you to create multiple vehicles easily and use APIs to control them.

Creating Multiple Vehicles

It's as easy as specifying them in settings.json. The `Vehicles` element allows you to specify list of vehicles you want to create along with their initial positions and orientations. The positions are specified in NED coordinates in SI units with origin set at Player Start component in Unreal environment. The orientation is specified as Yaw, Pitch and Roll in degrees.

Creating Multiple Cars

```
{
    "SettingsVersion": 1.2,
    "SimMode": "Car",

    "Vehicles": {
        "Car1": {
          "VehicleType": "PhysXCar",
          "X": 4, "Y": 0, "Z": -2
        },
        "Car2": {
          "VehicleType": "PhysXCar",
          "X": -4, "Y": 0, "Z": -2,
      "Yaw": 90
        }
  }
}
```

Creating Multiple Drones

```
{
    "SettingsVersion": 1.2,
    "SimMode": "Multirotor",

    "Vehicles": {
        "Drone1": {
          "VehicleType": "SimpleFlight",
          "X": 4, "Y": 0, "Z": -2,
      "Yaw": -180
        },
        "Drone2": {
          "VehicleType": "SimpleFlight",
          "X": 8, "Y": 0, "Z": -2
        }

    }
}
```

Using APIs for Multiple Vehicles

The new APIs since AirSim 1.2 allows you to specify `vehicle_name`. This name corresponds to keys in json settings (for example, Car1 or Drone2 above).

Example code for cars

Example code for multirotors

Demo

2.3.27 pfm Format

Pfm (or Portable FloatMap) image format stores image as floating point pixels and hence are not restricted to usual 0-255 pixel value range. This is useful for HDR images or images that describes something other than colors like depth.

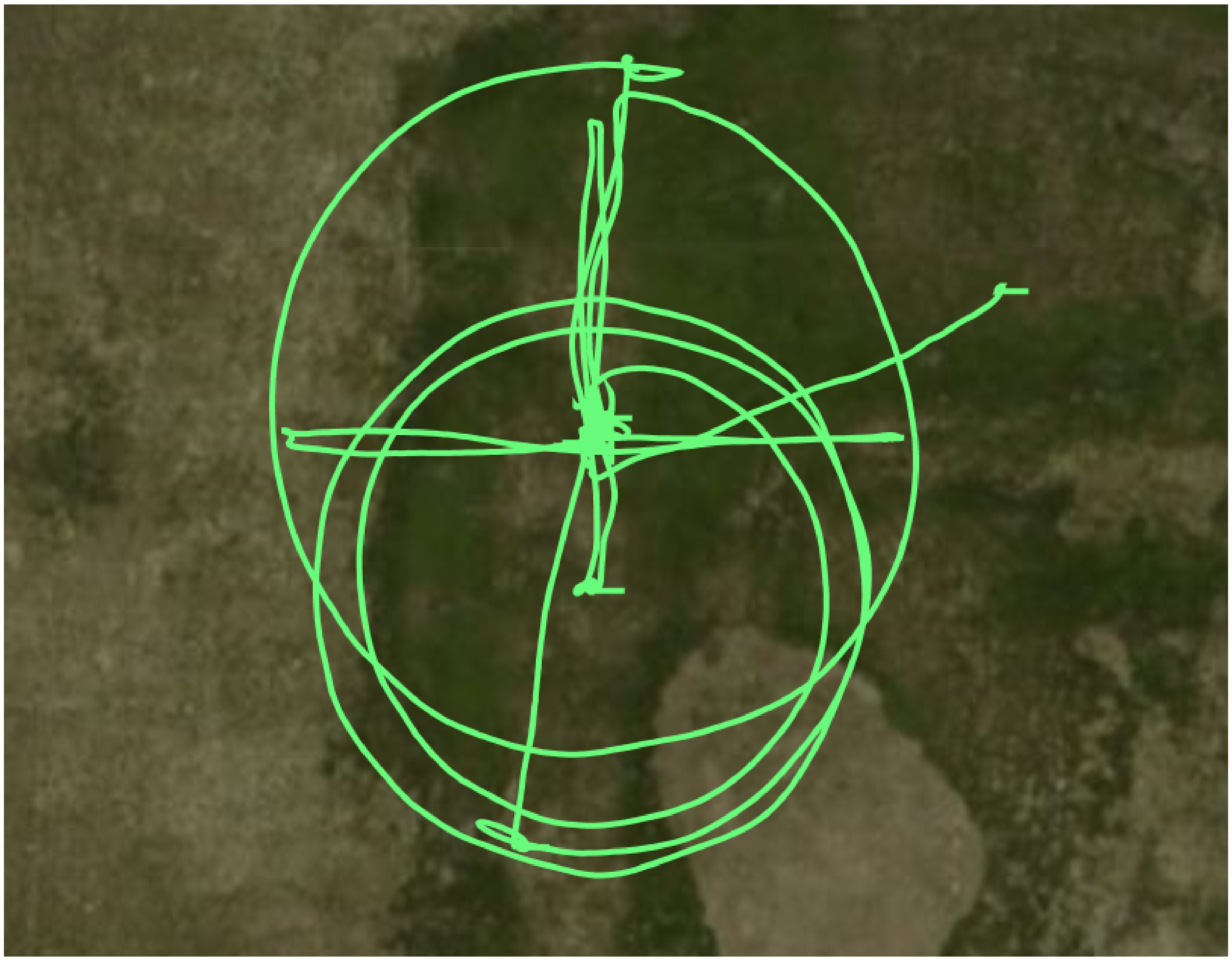
One of the good viewer to view this file format is PfmPad. We don't recommend Maverick photo viewer because it doesn't seem to show depth images properly.

AirSim has code to write pfm file for C++ and read as well as write for Python.

2.3.28 Playback

AirSim supports playing back the high level commands in a *.mavlink log file that were recorded using the MavLink-Test app for the purpose of comparing real and simulated flight. The recording.mavlink is an example of a log file captured using a real drone using the following command line:

Then the log file contains the commands performed, which included several "orbit" commands, the resulting GPS map of the flight looks like this:

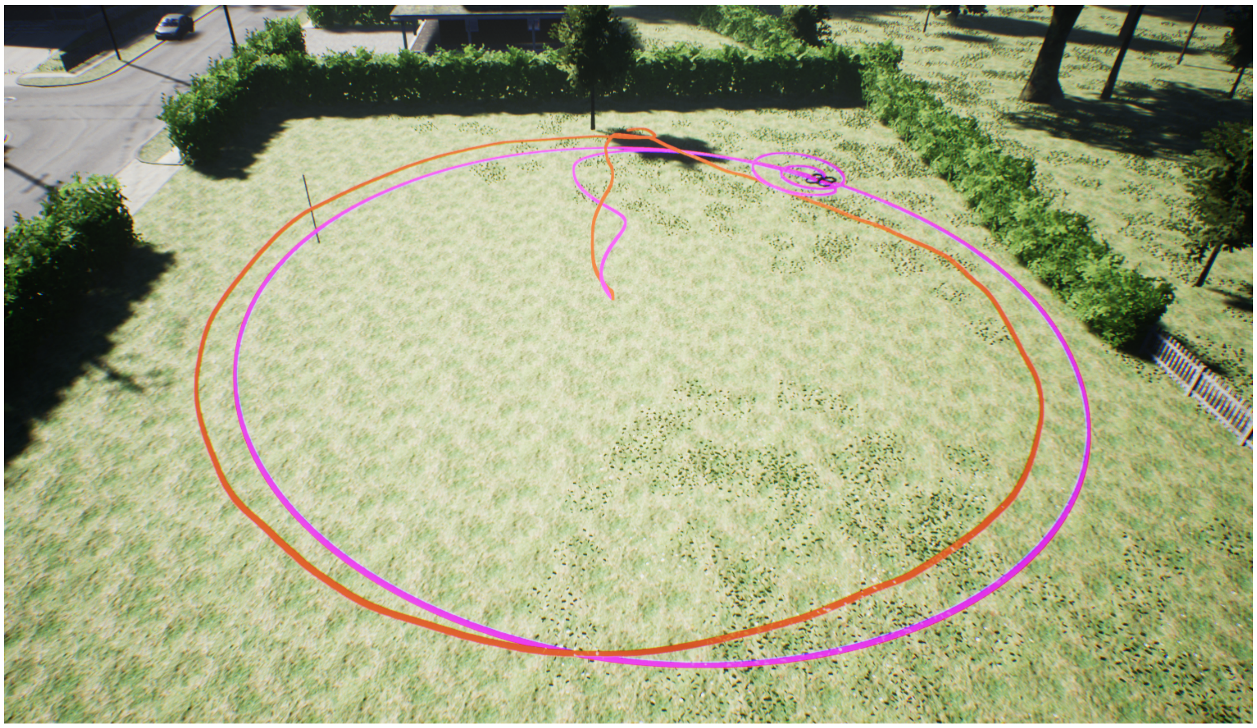


Side-by-side comparison

Now we can copy the *.mavlink log file recorded by MavLinkTest to the PC running the Unreal simulator with AirSim plugin. When the Simulator is running and the drone is parked in a place in a map that has room to do the same maneuvers we can run this MavLinkTest command line:

This should connect to the simulator. Now you can enter this command:

The same commands you performed on the real drone will now play again in the simulator. You can then press 't' to see the trace, and it will show you the trace of the real drone and the simulated drone. Every time you press 't' again you can reset the lines so they are sync'd to the current position, this way I was able to capture a side-by-side trace of the "orbit" command performed in this recording, which generates the picture below. The pink line is the simulated flight and the red line is the real flight:



Note: I'm using the ';' key in the simulator to take control of camera position using keyboard to get this shot.

Parameters

It may help to set the simulator up with some of the same flight parameters that your real drone is using, for example, in my case I was using a lower than normal cruise speed, slow takeoff speed, and it helps to tell the simulator to wait a long time before disarming (COM_DISARM_LAND) and to turn off the safety switches NAV_RCL_ACT and NAV_DLL_ACT (`don't` do that on a real drone).

2.3.29 Building PX4

Source code

Getting the PX4 source code is easy:

Oh, and if you don't have git yet just run this:

We are currently testing using the 1.6.0rc1 version, but the latest master branch should be ok too. Now to build it you will need the right tools.

PX4 Build tools

The full instructions are available on the dev.px4.io website, but we've copied the relevant subset of those instructions here for your convenience.

(Note that BashOnWindows) can be used to build the SITL version, but not the ARM firmware for pixhawk hardware).

First run this command to cache your admin credentials:

Now you can block copy/paste the following to a bash terminal and it should run them all to completion, but be sure to check each command for success:

Build SITL version

Now you can make the SITL version that runs in posix, from the Firmware folder you created above:

Note: this build system is quite special, it knows how to update git submodules (and there's a lot of them), then it runs cmake (if necessary), then it runs the build itself. So in a way the root Makefile is a meta-meta makefile :-)

It shouldn't take long, about 2 minutes. If all succeeds, the last line will link the `px4` app, which you can then run using the following:

And you should see output that looks like this:

so this is good, first run sets up the px4 parameters for SITL mode. Second run has less output. This app is also an interactive console where you can type commands. Type 'help' to see what they are and just type ctrl-C to kill it. You can do that and restart it any time, that's a great way to reset any wonky state if you need to (it's equivalent to a Pixhawk hardware reboot).

ARM embedded tools

If you plan to build the PX4 firmware for real Pixhawk hardware then you will need the gcc cross-compiler for ARM Cortex-M4 chipset. You can find out what version, if any, you may already have by typing this command `arm-none-eabi-gcc --version`. Note: you do not need this to build the SITL version of PX4.

Note: This does not work in BashOnWindows because the arm-none-eabi-gcc tool is a 32-bit app which BashOnWindows cannot run. See *installing arm-none-eabi-gcc on BashOnWindows* at the bottom of this page.

Anyway, first we make sure to remove any old version of arm-none-eabi-gcc:

That previous command prompts you to hit ENTER, so be sure to run that separately before the following:

So now when you type this command `arm-none-eabi-gcc --version` and you should see:

Build PX4 for ARM hardware

Now you can build the PX4 firmware for running on real pixhawk hardware:

This build will take a little longer because it is building a lot more including the NuttX real time OS, all the drivers for the sensors in the Pixhawk flight controller, and more. It is also running the compiler in super size-squeezing mode so it can fit all that in a 1 megabyte ROM !!

One nice tid bit is you can plug in your pixhawk USB, and type `make px4fmu-v2_default upload` to flash the hardware with these brand new bits, so you don't need to use QGroundControl for that.

Some Useful Parameters

PX4 has many customizable parameters (over 700 of them, in fact) and to get best results with AirSim we have found the following parameters are handy:

Installing arm-none-eabi-gcc in BashOnWindows

`SolinGuo` built a 64 bit version of `gcc-arm-none-eabi` so that it will run inside `BashOnWindows`. See gcc-arm-none-eabi-5_4-2017q2-20170512-linux.tar.bz2. If you download the *.tar.bz2 file to your machine and unpack it using this command line in BashOnWindows console:

you will get the following folder which contains the arm gcc cross-compiler:

If you add this folder to your PATH using the usual `export PATH=...` trick then the PX4 build will be able to find and run this compiler. After that, you can run `make px4fmu-v2_default` in BashOnWindows and the firmware will appear here: `build_px4fmu-v2_default/src/firmware/nuttx/px4fmu-v2_default.px4`. You can then flash this new firmware on your Pixhawk using QGroundControl.

2.3.30 PX4/MavLink Logging

Thanks to Chris Lovett for developing various tools for PX4/MavLink logging mentioned on this page!

Logging MavLink Messages

The following command will connect MavLinkTest app to the Simulator and enable logging of all mavlink commands to and from the PX4.

Sometimes you will also want to log the "output" from the Simulator that is being sent to the PX4. Specifically this will capture the "hilgps" and "hilsensor" messages that are generated by the Simulator. To do that run this as well:

You will then see log files organized by date in d:templogs, specifically *input.mavlink and *output.mavlink files.

MavLink LogViewer

For MavLink enabled drones, you can also use our Log Viewer to visualize the streams of data.

PX4 Log in SITL Mode

In SITL mode, please a log file is produced when drone is armed. The SITL terminal will contain the path to the log file, it should look something like this

PX4 Log in SITL Mode

If you are using Pixhawk hardware in HIL mode, then set parameter `SYS_LOGGER=1` using QGroundControl. PX4 will write log file on device which you can download at later date using QGroundControl.

2.3.31 PX4 Setup for AirSim

The PX4 software stack is an open source very popular flight controller with support for wide variety of boards and sensors as well as built-in capability for higher level tasks such as mission planning. Please visit px4.io for more information.

Warning: While all releases of AirSim are always tested with PX4 to ensure the support, setting up PX4 is not a trivial task. Unless you have at least intermediate level of experience with PX4 stack, we recommend you use simple_flight, which is now a default in AirSim.

Supported Hardware

The following Pixhawk hardware has been tested with AirSim:

1. 3DR Pixhawk v2
2. 3DR Pixhawk mini
3. Pixhawk PX4 2.4.8
4. PixFalcon
5. PixRacer
6. Pixhawk 2.1 (using PX4 Flight Stack)

The 3DR Pixhawk Mini works out of the box, the others you may need to re-flash with the latest firmware.

Setting up PX4 Hardware-in-Loop

For this you will need one of the supported device listed above. For manual flight you will also need RC + transmitter.

1. Make sure your RC receiver is bound with its RC transmitter. Connect the RC transmitter to the flight controller's RC port. Refer to your RC manual and PX4 docs for more information.
2. Download QGroundControl, launch it and connect your flight controller to the USB port.
3. Use QGroundControl to flash the latest PX4 Flight Stack. See also initial firmware setup video.
4. In QGroundControl, configure your Pixhawk for HIL simulation by selecting the HIL Quadrocopter X airframe. After PX4 reboots, check that "HIL Quadrocopter X" is indeed selected.
5. In QGroundControl, go to Radio tab and calibrate (make sure the remote control is on and the receiver is showing the indicator for the binding).
6. Go to the Flight Mode tab and chose one of the remote control switches as "Mode Channel". Then set (for example) Stabilized and Attitude flight modes for two positions of the switch.
7. Go to the Tuning section of QGroundControl and set appropriate values. For example, for Fly Sky's FS-TH9X remote control, the following settings give a more realistic feel: Hover Throttle = mid+1 mark, Roll and pitch sensitivity = mid-3 mark, Altitude and position control sensitivity = mid-2 mark.
8. In AirSim settings file, specify PX4 for your vehicle config like this: .. code-block:

```
{
"SettingsVersion": 1.2,
"SimMode": "Multirotor",
"Vehicles": {
 "PX4": {
   "VehicleType": "PX4Multirotor"
 }
}
}
```

After above setup you should be able to use RC to fly in the AirSim. You can usually arm the vehicle by lowering and bringing two sticks of RC together in-wards. You don't need QGroundControl after the initial setup. Typically the Stabilized (instead of Manual) mode gives better experience for beginners.

You can also control the drone from Python APIs.

See Walkthrough Demo Video and Unreal AirSim Setup Video that shows you all the setup steps in this document.

Setting up PX4 Software-in-Loop

The PX4 SITL mode doesn't require you to have separate device such as a Pixhawk or Pixracer. This is in fact the recommended way to use PX4 with simulators by PX4 team. However, this is indeed harder to set up. Please see this dedicated page for setting up PX4 in SITL mode.

FAQ

Drone doesn't fly properly, it just goes "crazy".

There are a few reasons that can cause this. First, make sure your drone doesn't fall down large distance when starting the simulator. This might happen if you have created a custom Unreal environment and Player Start is placed too high above the ground. It seems that when this happens internal calibration in PX4 gets confused.

You should also use QGroundControl and make sure you can arm and takeoff in QGroundControl properly.

Finally, this also can be a machine performance issue in some rare cases, check your hard drive performance.

Can I use Arducopter or other MavLink implementations?

Our code is tested with the PX4 firmware. We have not tested Arducopter or other mavlink implementations. Some of the flight API's do use the PX4 custom modes in the MAV_CMD_DO_SET_MODE messages (like PX4_CUSTOM_MAIN_MODE_AUTO)

It is not finding my Pixhawk hardware

Check your settings.json file for this line "SerialPort":"*,115200". The asterisk here means "find any serial port that looks like a Pixhawk device, but this doesn't always work for all types of Pixhawk hardware. So on Windows you can find the actual COM port using Device Manager, look under "Ports (COM & LPT), plug the device in and see what new COM port shows up. Let's say you see a new port named "USB Serial Port (COM5)". Well, then change the SerialPort setting to this: "SerialPort":"COM5,115200".

On Linux, the device can be found by running "ls /dev/serial/by-id" if you see a device name listed that looks like this `usb-3D_Robotics_PX4_FMU_v2.x_0-if00` then you can use that name to connect, like this: "SerialPort":"/dev/serial/by-id/usb-3D_Robotics_PX4_FMU_v2.x_0-if00". Note this long name is actually a symbolic link to the real name, if you use "ls -l ..." you can find that symbolic link, it is usually something like "/dev/ttyACM0", so this will also work "SerialPort":"/dev/ttyACM0,115200". But that mapping is similar to windows, it is automatically assigned and can change, whereas the long name will work even if the actual TTY serial device mapping changes.

WARN [commander] Takeoff denied, disarm and re-try

This happens if you try and take off when PX4 still has not computed the home position. PX4 will report the home position once it is happy with the GPS signal, and you will see these messages:

Up until this point in time, however, the PX4 will reject takeoff commands.

When I tell the drone to do something it always lands

For example, you use DroneShell `moveToPosition -z -20 -x 50 -y 0` which it does, but when it gets to the target location the drone starts to land. This is the default behavior of PX4 when offboard mode completes. To set the drone to hover instead set this PX4 parameter:

I get message length mismatches errors

You might need to set MAV_PROTO_VER parameter in QGC to "Always use version 1". Please see this issue more details.

2.3.32 Setting up PX4 Software-in-Loop

The PX4 software provides a "software-in-loop" simulation (SITL) version of their stack that runs in Linux. Sorry it doesn't run in Windows, but if you install BashOnWindows you can build and run it there.

1. From your Linux bash terminal follow these steps for Linux and follow **all** the instructions under `NuttX based hardware` to install prerequisites. We've also included out own copy of the PX4 build instructions which is a bit more concise about what we need exactly.
2. Get the PX4 source code and build the posix SITL version of PX4:
3. Use following command to start PX4 firmware in SITL mode: .. code-block:

```
./build/posix_sitl_default/px4 ./posix-configs/SITL/init/ekf2/iris
```

4. You should see a message like this you `INFO [simulator] Waiting for initial data on UDP port 14560` which means the SITL PX4 app is waiting for someone to connect.
5. Now edit AirSim settings file to make sure you have followings: .. code-block:: json

   { "SettingsVersion": 1.2, "SimMode": "Multirotor", "Vehicles": {

   "PX4": { "VehicleType": "PX4Multirotor", "UseSerial": false

   }

   }

   }

6. Run Unreal environment and it should connect to SITL via UDP. You should see a bunch of messages from the SITL PX4 window from things like `[mavlink]` and `[commander]` and so on.
7. You should also be able to use QGroundControl just like with flight controller hardware. Note that as we don't have physical board, RC cannot be connected directly to it. So the alternatives are either use XBox 360 Controller or connect your RC using USB (for example, in case of FrSky Taranis X9D Plus) or using trainer USB cable to PC. This makes your RC look like joystick. You will need to do extra set up in QGroundControl to use virtual joystick for RC control.

Setting GPS origin

PX4 SITL mode needs to be configured to get the home location correct. Run the following in the PX4 console window so that the origin matches that which is setup in AirSim AVehiclePawnBase::HomeLatitude and HomeLongitude.

You might also want to set this one so that the drone automatically hovers after each offboard control command (the default setting is to land):

Now close Unreal app, restart `./build_posix_sitl_default/src/firmware/posix/px4` and re-start the unreal app.

Check the Home Position

If you are using DroneShell to execute commands (arm, takeoff, etc) then you should wait until the Home position is set. You will see the PX4 SITL console output this message:

Now DroneShell 'pos' command should report this position and the commands should be accepted by PX4. If you attempt to takeoff without a home position you will see the message:

After home position is set check the local position reported by 'pos' command :

If the z coordinate is large like this then takeoff might not work as expected. Resetting the SITL and simulation should fix that problem.

No Remote Control

If you plan to fly with no remote control, just using DroneShell commands for example, then you will need to set the following parameters to stop the PX4 from triggering "failsafe mode on" every time a move command is finished.

NOTE: Do `NOT` do this on a real drone as it is too dangerous to fly without these failsafe measures.

Using VirtualBox Ubuntu

If you want to run the above posix_sitl in a `VirtualBox Ubuntu` machine then it will have a different ip address from localhost. So in this case you need to edit the settings file and change the UdpIp and SitlIp to the ip address of your virtual machine set the LocalIpAddress to the address of your host machine running the Unreal engine.

Remote Controller

There are several options for flying the simulated drone using a remote control or joystick like xbox gamepad. See remote controllers

2.3.33 Using Python APIs for AirSim

This document is now merged with general APIs document.

2.3.34 Reinforcement Learning in AirSim

We below describe how we can implement DQN in AirSim using CNTK. The easiest way is to first install python only CNTK (instructions).

CNTK provides several demo examples of deep RL. We will modify the DeepQNeuralNetwork.py to work with AirSim. We can utilize most of the classes and methods corresponding to the DQN algorithm. However, there are certain additions we need to make for AirSim.

Disclaimer

This is still in active development. What we share below is a framework that can be extended and tweaked to obtain better performance.

RL with Car

Source code

This example works with AirSimNeighborhood environment available in releases.

First, we need to get the images from simulation and transform them appropriately. Below, we show how a depth image can be obtained from the ego camera and transformed to an 84X84 input to the network. (you can use other sensor modalities, and sensor inputs as well – of course you'll have to modify the code accordingly).

We further define the six actions (breaking, straight with throttle, full-left with throttle, full-right with throttle, half-left with throttle, half-right with throttle) that an agent can execute. This is done via the function `interpret_action`:

We then define the reward function in `compute_reward` as a convex combination of how fast the vehicle is travelling and how much it deviates from the center line. The agent gets a high reward when its moving fast and staying in the center of the lane.

The function `isDone` determines if the episode has terminated (e.g. due to collision). We look at the speed of the vehicle and if it is less than a threshold than the episode is considered to be terminated.

The main loop then sequences through obtaining the image, computing the action to take according to the current policy, getting a reward and so forth. If the episode terminates then we reset the vehicle to the original state via:

Note that the simulation needs to be up and running before you execute DQNcar.py. The video below shows first few episodes of DQN training.

RL with Quadrotor

Source code

This example works with AirSimMountainLandscape environment available in releases.

We can similarly apply RL for various autonomous flight scenarios with quadrotors. Below is an example on how RL could be used to train quadrotors to follow high tension power lines (e.g. application for energy infrastructure inspection). There are seven actions here that correspond to different directions in which the quadrotor can move in (six directions + one hovering action).

The reward again is a function how how fast the quad travels in conjunction with how far it gets from the known powerlines.

We consider an episode to terminate if it drifts too much away from the known power line coordinates.

The reset function here flies the quadrotor to the initial starting point:

Here is the video of first few episodes during the training.

Related

Please also see The Autonomous Driving Cookbook by Microsoft Deep Learning and Robotics Garage Chapter.

2.3.35 Release Notes

Please see 'What's new <whats_new.md>'_. This page is not maintained currently.

v1.1 - 2017-09-28

- We have added car model!

v1.0 - 2017-08-29

- simple_flight is now default flight controller for drones. If you want to use PX4, you will need to modify settings.json as per PX4 setup doc.
- Linux build is official and currently uses Unreal 4.17 due to various bug fixes required
- ImageType enum has breaking changes with several new additions and clarifying existing ones
- SubWindows are now configurable from settings.json
- PythonClient is now complete and has parity with C++ APIs. Some of these would have breaking changes.

2.3.36 Remote Control

To fly manually, you need remote control or RC. If you don't have one then you can use APIs to fly programmatically or use so-called Computer Vision mode to move around using keyboard.

RC Setup for Default Config

By default AirSim uses simple_flight as its flight controller which connects to RC via USB port to your computer.

You can either use XBox controller or FrSky Taranis X9D Plus. Note that XBox 360 controller is not precise enough and is not recommended if you wanted more real world experience. See FAQ below if things are not working.

Other Devices

AirSim can detect large variety of devices however devices other than above *might* need extra configuration. In future we will add ability to set this config through settings.json. For now, if things are not working then you might want to try workarounds such as x360ce or chnage code in SimJoystick.cpp file.

Note on FrSky Taranis X9D Plus

FrSky Taranis X9D Plus is real UAV remote control with an advantage that it has USB port so it can be directly connected to PC. You can download AirSim config file and follow this tutorial to import it in your RC. You should then see "sim" model in RC with all channels configured properly.

Note on Linux

Currently default config on Linux is for using Xbox controller. This means other devices might not work properly. In future we will add ability to configure RC in settings.json but for now you *might* have to change code in SimJoystick.cpp file to use other devices.

RC Setup for PX4

AirSim supports PX4 flight controller however it requires different setup. There are many remote control options that you can use with quadrotors. We have successfully used FrSky Taranis X9D Plus, FlySky FS-TH9X and Futaba 14SG with AirSim. Following are the high level steps to configure your RC:

1. If you are going to use Hardware-in-Loop mode, you need transmitter for your specific brand of RC and bind it. You can find this information in RC's user guide.
2. For Hardware-in-Loop mode, you connect transmitter to Pixhawk. Usually you can find online doc or YouTube video tutorial on how to do that.
3. Calibrate your RC in QGroundControl.

See PX4 RC configuration and Please see this guide for more information.

Using XBox 360 USB Gamepad

You can also use an xbox controller in SITL mode, it just won't be as precise as a real RC controller. See xbox controller for details on how to set that up.

Using Playstation 3 controller

A Playstation 3 controller is confirmed to work as an AirSim controller. On Windows, an emulator to make it look like an Xbox 360 controller, is required however. Many different solutions are available online, for example x360ce Xbox 360 Controller Emulator.

DJI Controller

Nils Tijtgat wrote an excellent blog on how to get the DJI controller working with AirSim.

FAQ

This typically happens if you have multiple RCs and or XBox/Playstation gamepads etc connected. In Windows, hit Windows+S key and search for "Set up USB Game controllers" (in older versions of Windows try "joystick"). This will show you all game controllers connected to your PC. If you don't see yours than Windows haven't detected it and so you need to first solve that issue. If you do see yours but not at the top of the list (i.e. index 0) than you need to tell AirSim because AirSim by default tries to use RC at index 0. To do this, navigate to your `~/Documents/AirSim` folder, open up `settings.json` and add/modify following setting. Below tells AirSim to use RC at index = 2.

Regular gamepads are not very precise and have lot of random noise. Most of the times you may see significant offsets as well (i.e. output is not zero when sticks are at zero). So this behavior is expected.

We haven't implemented it yet. This means your RC firmware will need to have a capability to do calibration for now.

First you want to make sure your RC is working in QGroundControl. If it doesn't then it will sure not work in AirSim. The PX4 mode is suitable for folks who have at least intermediate level of experience to deal with various issues related to PX4 and we would generally refer you to get help from PX4 forums.

2.3.37 How to use AirSim with Robot Operating System (ROS)

AirSim and ROS can be integrated using C++ or Python. Some example ROS nodes are provided demonstrating how to publish data from AirSim as ROS topics.

2.3.38 Python

Prerequisites

These instructions are for Ubuntu 16.04, ROS Kinetic, UE4 4.18 or higher, and latest AirSim release. You should have these components installed and working before proceeding

Setup

Create a new ROS package in your catkin workspace following these instructions.

Create a new ROS package called airsim or whatever you like.

Create ROS package

If you don't already have a catkin workspace, you should first work through the ROS beginner tutorials.

Add AirSim ROS node examples to ROS package

In the ROS package directory you made, copy the ros examples from the AirSim/PythonClient directory to your ROS package. Change the code below to match your AirSim and catkin workspace paths.

Build ROS AirSim package

Change directory to your top level catkin workspace folder i.e. `cd ~/catkin_ws` and run `catkin_make` This will build the airsim package. Next, run `source devel/setup.bash` so ROS can find the new package. You can add this command to your ~/.bashrc to load your catkin workspace automatically.

How to run ROS AirSim nodes

First make sure UE4 is running an airsim project, the car or drone should be selected, and the simulations is playing. Examples support car or drone. Make sure to have the correct vehicle for the ros example running.

The example airsim nodes can be run using `rosrun airsim example_name.py` The output of the node can be viewed in another terminal by running `rostopic echo /example_name` You can view a list of the topics currently published via tab completion after typing `rostopic echo` in the terminal. Rviz is a useful visualization tool that can display the published data.

2.3.39 C++ (coming soon)

2.3.40 Sensors in AirSim

AirSim currently supports the following sensors:

- Camera
- Imu
- Magnetometer
- Gps
- Barometer
- Distance
- Lidar

The cameras are currently configured a bit differently than other sensors. The camera configuration and apis are covered in other documents, e.g., general settings and image API.

This document focuses on the configuration of other sensors.

Default sensors

If not sensors are specified in the settings json, the the following sensors are enabled by default based on the simmode.

Multirotor

- Imu
- Magnetometer
- Gps
- Barometer ### Car
- Gps ### ComputerVision
- None

Please see ‘createDefaultSensorSettings’ method in AirSimSettings.hpp

Configuration of Default Sensor list

A default sensor list can be configured in settings json. e.g.,

Configuration of vehicle specific sensor settings

A vehicle specific sensor list can be specified in the vehicle settings part of the json. e.g.,

If a vehicle provides its sensor list, it must provide the whole list. Selective add/remove/update of the default sensor list is NOT supported.

Configuration of sensor settings

Shared settings

There are two shared settings:

- SensorType .. code-block:

```
An integer representing the sensor-type [SensorBase.hpp](../AirLib/include/
↪sensors/)
```

- **Enabled** Boolean

Sensor specific settings

Each sensor-type has its own set of settings as well. Please see lidar for example of Lidar specific settings.

Sensor APIs

Each sensor-type has its own set of APIs currently. Please see lidar for example of Lidar specific APIs.

2.3.41 AirSim Settings

Where are Settings Stored?

Windows: `Documents\AirSim` Linux: `~/Documents/AirSim`

The file is in usual json format. On first startup AirSim would create `settings.json` file with no settings. To avoid problems, always use ASCII format to save json file.

How to Chose Between Car and Multirotor?

The default is to use multirotor. To use car simple set `"SimMode": "Car"` like this:

To choose multirotor, set `"SimMode": "Multirotor"`. If you want to prompt user to select vehicle type then use `"SimMode": ""`.

Available Settings and Their Defaults

Below are complete list of settings available along with their default values. If any of the settings is missing from json file, then default value is used. Some default values are simply specified as `""` which means actual value may be chosen based on the vehicle you are using. For example, `ViewMode` setting has default value `""` which translates to `"FlyWithMe"` for drones and `"SpringArmChase"` for cars.

WARNING: Do not copy paste all of below in your settings.json. We strongly recommend adding only those settings that you don't want default values. Only required element is `"SettingsVersion"`.

SimMode

SimMode determines which simulation mode will be used. Below are currently supported values:

- `""`: prompt user to select vehicle type multirotor or car
- `"Multirotor"`: Use multirotor simulation
- `"Car"`: Use car simulation
- `"ComputerVision"`: Use only camera, no vehicle or physics

ViewMode

The ViewMode determines which camera to use as default and how camera will follow the vehicle. For multirotors, the default ViewMode is `"FlyWithMe"` while for cars the default ViewMode is `"SpringArmChase"`.

- `FlyWithMe`: Chase the vehicle from behind with 6 degrees of freedom
- `GroundObserver`: Chase the vehicle from 6' above the ground but with full freedom in XY plane.
- `Fpv`: View the scene from front camera of vehicle
- `Manual`: Don't move camera automatically. Use arrow keys and ASWD keys for move camera manually.
- `SpringArmChase`: Chase the vehicle with camera mounted on (invisible) arm that is attached to the vehicle via spring (so it has some latency in movement).
- `NoDisplay`: This will freeze rendering for main screen however rendering for subwindows, recording and APIs remain active. This mode is useful to save resources in "headless" mode where you are only interested in getting images and don't care about what gets rendered on main screen. This may also improve FPS for recording images.

TimeOfDay

This setting controls the position of Sun in the environment. By default `Enabled` is false which means Sun's position is left at whatever was the default in the environment and it doesn't change over the time. If `Enabled` is true then Sun position is computed using longitude, latitude and altitude specified in `OriginGeopoint` section for the date specified in `StartDateTime` in the string format as `%Y-%m-%d %H:%M:%S`, for example, `2018-02-12 15:20:00`. If this string is empty then current date and time is used. If `StartDateTimeDst` is true then we adjust for day light savings time. The Sun's position is then continuously updated at the interval specified in `UpdateIntervalSecs`. In some cases, it might be desirable to have celestial clock run faster or slower than simulation clock. This can be specified using `CelestialClockSpeed`, for example, value 100 means for every 1 second of simulation clock, Sun's position is advanced by 100 seconds so Sun will move in sky much faster.

OriginGeopoint

This setting specifies the latitude, longitude and altitude of the Player Start component placed in the Unreal environment. The vehicle's home point is computed using this transformation. Note that all coordinates exposed via APIs are using NED system in SI units which means each vehicle starts at (0, 0, 0) in NED system. Time of Day settings are computed for geographical coordinates specified in `OriginGeopoint`.

SubWindows

This setting determines what is shown in each of 3 subwindows which are visible when you press 0 key. The WindowsID can be 0 to 2, CameraName is any available camera on the vehicle. ImageType integer value determines what kind of image gets shown according to ImageType enum. For example, for car vehicles below shows driver view, front bumper view and rear view as scene, depth and surface normals respectively.

Recording

The recording feature allows you to record data such as position, orientation, velocity along with the captured image at specified intervals. You can start recording by pressing red Record button on lower right or the R key. The data is stored in the `Documents\AirSim` folder, in a time stamped subfolder for each recording session, as tab separated file.

- `RecordInterval`: specifies minimal interval in seconds between capturing two images.
- `RecordOnMove`: specifies that do not record frame if there was vehicle's position or orientation hasn't changed.
- `Cameras`: this element controls which cameras are used to capture images. By default scene image from camera 0 is recorded as compressed png format. This setting is json array so you can specify multiple cameras to capture images, each with potentially different image types. When PixelsAsFloat is true, image is saved as pfm file instead of png file.

ClockSpeed

This setting allows you to set the speed of simulation clock with respect to wall clock. For example, value of 5.0 would mean simulation clock has 5 seconds elapsed when wall clock has 1 second elapsed (i.e. simulation is running faster). The value of 0.1 means that simulation clock is 10X slower than wall clock. The value of 1 means simulation is running in real time. It is important to realize that quality of simulation may decrease as the simulation clock runs faster. You might see artifacts like object moving past obstacles because collision is not detected. However slowing down simulation clock (i.e. values < 1.0) generally improves the quality of simulation.

Segmentation Settings

The `InitMethod` determines how object IDs are initialized at startup to generate segmentation. The value "" or "CommonObjectsRandomIDs" (default) means assign random IDs to each object at startup. This will generate segmentation view with random colors assign to each object. The value "None" means don't initialize object IDs. This will cause segmentation view to have single solid colors. This mode is useful if you plan to set up object IDs using APIs and it can save lot of delay at startup for large environments like CityEnviron.

> If `OverrideExisting` is false then initialization does not alter non-zero object IDs already assigned otherwise it does.
>
> If `MeshNamingMethod` is "" or "OwnerName" then we use mesh's owner name to generate random hash as object IDs. If its "StaticMeshName" then we use static mesh's name to generate random hash as object IDs. Note that it is not possible to tell individual instances of the same static mesh apart this way, but the names are often more intuitive.

Camera Settings

The `CameraDefaults` element at root level specifies defaults used for all cameras. These defaults can be overridden for individual camera in `Cameras` element inside `Vehicles` as described later.

Note on ImageType element

The ImageType element in JSON array determines which image type that settings applies to. The valid values are described in ImageType section. In addition, we also support special value ImageType: -1 to apply the settings to external camera (i.e. what you are looking at on the screen).

For example, CaptureSettings element is json array so you can add settings for multiple image types easily.

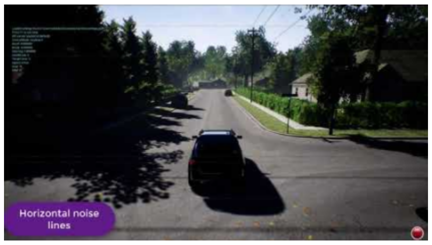
CaptureSettings

The CaptureSettings determines how different image types such as scene, depth, disparity, surface normals and segmentation views are rendered. The Width, Height and FOV settings should be self explanatory. The AutoExposureSpeed decides how fast eye adaptation works. We set to generally high value such as 100 to avoid artifacts in image capture. Similarly we set MotionBlurAmount to 0 by default to avoid artifacts in ground truth images. The ProjectionMode decides the projection used by the capture camera and can take value "perspective" (default) or "orthographic". If projection mode is "orthographic" then OrthoWidth determines width of projected area captured in meters.

For explanation of other settings, please see this article.

NoiseSettings

The NoiseSettings allows to add noise to the specified image type with a goal of simulating camera sensor noise, interference and other artifacts. By default no noise is added, i.e., Enabled: false. If you set Enabled: true then following different types of noise and interference artifacts are enabled, each can be further tuned using setting. The noise effects are implemented as shader created as post processing material in Unreal Engine called CameraSensorNoise.

Demo of camera noise and interference simulation: 

Random noise

This adds random noise blobs with following parameters.

- RandContrib: This determines blend ratio of noise pixel with image pixel, 0 means no noise and 1 means only noise.
- RandSpeed: This determines how fast noise fluctuates, 1 means no fluctuation and higher values like 1E6 means full fluctuation.
- RandSize: This determines how coarse noise is, 1 means every pixel has its own noise while higher value means more than 1 pixels share same noise value.
- RandDensity: This determines how many pixels out of total will have noise, 1 means all pixels while higher value means lesser number of pixels (exponentially).

Horizontal bump distortion

This adds horizontal bumps / flickering / ghosting effect.

- `HorzWaveContrib`: This determines blend ratio of noise pixel with image pixel, 0 means no noise and 1 means only noise.
- `HorzWaveStrength`: This determines overall strength of the effect.
- `HorzWaveVertSize`: This determines how many vertical pixels would be effected by the effect.
- `HorzWaveScreenSize`: This determines how much of the screen is effected by the effect.

Horizontal noise lines

This adds regions of noise on horizontal lines.

- `HorzNoiseLinesContrib`: This determines blend ratio of noise pixel with image pixel, 0 means no noise and 1 means only noise.
- `HorzNoiseLinesDensityY`: This determines how many pixels in horizontal line gets affected.
- `HorzNoiseLinesDensityXY`: This determines how many lines on screen gets affected.

Horizontal line distortion

This adds fluctuations on horizontal line.

- `HorzDistortionContrib`: This determines blend ratio of noise pixel with image pixel, 0 means no noise and 1 means only noise.
- `HorzDistortionStrength`: This determines how large is the distortion.

Gimbal

The `Gimbal` element allows to freeze camera orientation for pitch, roll and/or yaw. This setting is ignored unless `ImageType` is -1. The `Stabilization` is defaulted to 0 meaning no gimbal i.e. camera orientation changes with body orientation on all axis. The value of 1 means full stabilization. The value between 0 to 1 acts as a weight for fixed angles specified (in degrees, in world-frame) in `Pitch`, `Roll` and `Yaw` elements and orientation of the vehicle body. When any of the angles is omitted from json or set to NaN, that angle is not stabilized (i.e. it moves along with vehicle body).

Vehicles Settings

Each simulation mode will go through the list of vehicles specified in this setting and create the ones that has `"AutoCreate": true`. Each vehicle specified in this setting has key which becomes the name of the vehicle. If `"Vehicles"` element is missing then this list is populated with default car named "PhysXCar" and default multirotor named "SimpleFlight".

Common Vehicle Setting

- `VehicleType`: This could be either `PhysXCar`, `SimpleFlight`, `PX4Multirotor` or `ComputerVision`. There is no default value therefore this element must be specified.

- `PawnPath`: This allows to override the pawn blueprint to use for the vehicle. For example, you may create new pawn blueprint derived from ACarPawn for a warehouse robot in your own project outside the AirSim code and then specify its path here. See also *PawnPaths*.
- `DefaultVehicleState`: Possible value for multirotors is `Armed` or `Disarmed`.
- `AutoCreate`: If true then this vehicle would be spawned (if supported by selected sim mode).
- `RC`: This sub-element allows to specify which remote controller to use for vehicle using `RemoteControlID`. The value of -1 means use keyboard (not supported yet for multirotors). The value >= 0 specifies one of many remote controllers connected to the system. The list of available RCs can be seen in Game Controllers panel in Windows, for example.
- `X, Y, Z, Yaw, Roll, Pitch`: These elements allows you to specify the initial position and orientation of the vehicle. Position is in NED coordinates in SI units with origin set to Player Start location in Unreal environment. The orientation is specified in degrees.
- `IsFpvVehicle`: This setting allows to specify which vehicle camera will follow and the view that will be shown when ViewMode is set to Fpv. By default, AirSim selects the first vehicle in settings as FPV vehicle.
- *Cameras*: This element specifies camera settings for vehicle. The key in this element is name of the [available camera](image_apis.md#available_cameras) and the value is same as *CameraDefaults* as described above. For example, to change FOV for the front center camera to 120 degrees, you can use this for *Vehicles* setting:

```
"Vehicles": {
    "FishEyeDrone": {
      "VehicleType": "SimpleFlight",
      "Cameras": {
        "front-center": {
          "CaptureSettings": [
            {
              "ImageType": 0,
              "FOV_Degrees": 120
            }
          ]
        }
      }
    }
}
```

Using PX4

By default we use simple_flight so you don't have to do separate HITL or SITL setups. We also support "PX4" for advanced users. To use PX4 with AirSim, you can use the following for `Vehicles` setting:

Additional PX4 Settings

The defaults for PX4 is to enable hardware-in-loop setup. There are various other settings available for PX4 as follows with their default values:

These settings define the MavLink SystemId and ComponentId for the Simulator (SimSysID, SimCompID), and for an optional external renderer (ExtRendererSysID, ExtRendererCompID) and the node that allows remote control of the drone from another app this is called the Air Control node (AirControlSysID, AirControlCompID).

If you want the simulator to also talk to your ground control app (like QGroundControl) you can also set the UDP address for that in case you want to run that on a different machine (QgcHostIp,QgcPort).

You can connect the simulator to the LogViewer app, provided in this repo, by setting the UDP address for that (LogViewerHostIp,LogViewerPort).

And for each flying drone added to the simulator there is a named block of additional settings. In the above you see the default name "PX4". You can change this name from the Unreal Editor when you add a new BP_FlyingPawn asset. You will see these properties grouped under the category "MavLink". The MavLink node for this pawn can be remote over UDP or it can be connected to a local serial port. If serial then set UseSerial to true, otherwise set UseSerial to false and set the appropriate bard rate. The default of 115200 works with Pixhawk version 2 over USB.

Other Settings

EngineSound

To turn off the engine sound use setting `"EngineSound": false`. Currently this setting applies only to car.

PawnPaths

This allows you to specify your own vehicle pawn blueprints, for example, you can replace the default car in AirSim with your own car. Your vehicle BP can reside in Content folder of your own Unreal project (i.e. outside of AirSim plugin folder). For example, if you have a car BP located in file `Content\MyCar\MySedanBP.uasset` in your project then you can set `"DefaultCar": {"PawnBP":"Class'/Game/MyCar/MySedanBP.MySedanBP_C'"}`. The `XYZ.XYZ_C` is a special notation required to specify class for BP `XYZ`. Please note that your BP must be derived from CarPawn class. By default this is not the case but you can re-parent the BP using the "Class Settings" button in toolbar in UE editor after you open the BP and then choosing "Car Pawn" for Parent Class settings in Class Options. It's also a good idea to disable "Auto Possess Player" and "Auto Possess AI" as well as set AI Controller Class to None in BP details. Please make sure your asset is included for cooking in packaging options if you are creating binary.

PhysicsEngineName

For cars, we support only PhysX for now (regardless of value in this setting). For multirotors, we support `"FastPhysicsEngine"` only.

LocalHostIp Setting

Now when connecting to remote machines you may need to pick a specific Ethernet adapter to reach those machines, for example, it might be over Ethernet or over Wi-Fi, or some other special virtual adapter or a VPN. Your PC may have multiple networks, and those networks might not be allowed to talk to each other, in which case the UDP messages from one network will not get through to the others.

So the LocalHostIp allows you to configure how you are reaching those machines. The default of 127.0.0.1 is not able to reach external machines, this default is only used when everything you are talking to is contained on a single PC.

SpeedUnitFactor

Unit conversion factor for speed related to `m/s`, default is 1. Used in conjunction with SpeedUnitLabel. This may be only used for display purposes for example on-display speed when car is being driven. For example, to get speed in `miles/hr` use factor 2.23694.

SpeedUnitLabel

Unit label for speed, default is `m/s`. Used in conjunction with SpeedUnitFactor.

2.3.42 simple_flight

If you don't know what flight controller does than see What is Flight Controller?.

AirSim has built-in flight controller called simple_flight and it is used by default. You don't need to do anything to use or configure it. AirSim also supports PX4 as another flight controller for advanced users. In future, we also plan to support ROSFlight and Hackflight.

Advantages

The advantage of using simple_flight is zero additional setup you need to do and it "just works". Also, simple_flight uses steppable clock which means you can pause the simulation and things are not at mercy of high variance low precision clock that operating system provides. Further, simple_flight is simple, cross platform and 100% header-only dependency-free C++ code which means you can literally step through from simulator to inside flight controller code within same code base!

Design

Normally flight controllers are designed to run on actual hardware on vehicles and their support for running in simulator varies widely. They are often fairly difficult to configure for non-expert users and typically have complex build usually lacking cross platform support. All these problems have played significant part in design of simple_flight.

simple_flight is designed from ground up as library with clean interface that can work onboard the vehicle as well as simulator. The core principle is that flight controller has no way to specify special simulation mode and there for it has no way to know if it is running under simulation or real vehicle. We thus view flight controller simply as collection of algorithms packaged in a library. Another key emphasis is to develop this code as dependency free header-only pure standard C++11 code. This means there is no special build required to compile simple_flight. You just copy its source code to any project you wish and it just works.

Control

simple_flight can control vehicle by taking in desired input as angle rate, angle level, velocity or position. Each axis of control can be specified with one of these modes. Internally simple_flight uses cascade of PID controllers to finally generate actuator signals. This means position PID drives velocity PID which drives angle level PID which finally drives angle rate PID.

State Estimation

In current release we are using ground truth from simulator for our state estimation. We plan to add complimentary filter based state estimation for angular velocity and orientation using 2 sensors (gyroscope, accelerometer) in near future. In more longer term, we plan to integrate another library to do velocity and position estimation using 4 sensors (gyroscope, accelerometer, magnetometer and barometer) using EKF. If you have experience this area than we encourage you to engage with us and contribute!

Supported Boards

Currently we have implemented simple_flight interfaces for simulated board. We plan to implement it for Pixhawk V2 board and possibly Naze32 board. We expect all our code to remain unchanged and the implementation would mainly involve adding drivers for various sensors, handling ISRs and managing other board specific details. If you have experience this area than we encourage you to engage with us and contribute!

Configuration

To have AirSim use simple_flight, you can specify it in settings.json as shown below. Note that this is default so you don't have to do it explicitly.

By default, vehicle using simple_flight is already armed which is why you would see propellers spinning. However if you don't want that than set `DefaultVehicleState` to `Inactive` like this:

In this case, you will need to either manually arm using RC sticks in down inward position or using APIs.

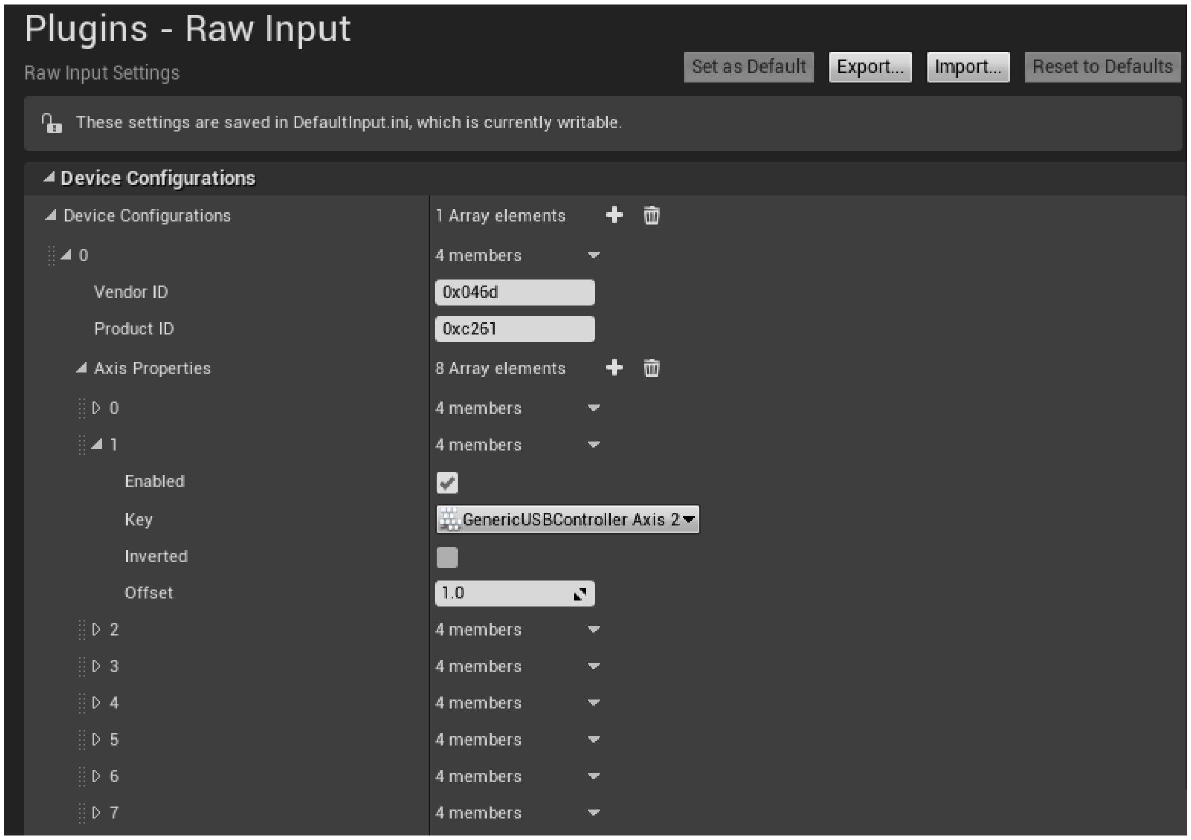
For safety reasons, flight controllers would disallow API control unless human operator has consented using a switch on RC. Also, when RC control is lost, vehicle should disable API control and enter hover mode for safety reasons. To simplify things a bit, simple_flight enables API control without human consent using RC and even when RC is not detected by default however you can change this using following setting:

Finally, simple_flight uses steppable clock by default which means clock advances when simulator tells it to advance (unlike wall clock which advances strictly according to passage of time). This means clock can be paused, for example, if code hits the break point and there is zero variance in clock (clock APIs provides by operating system might have significant variance unless its "real time" OS). If you want simple_flight to use wall clock instead than use following settings:

2.3.43 Logitech G920 Steering Wheel Installation

To use Logitech G920 steering wheel with AirSim follow these steps:

1. Connect the steering wheel to the computer and wait until drivers installation complete.
2. Install Logitech Gaming Software from here
3. Before debug, you'll have to normalize the values in AirSim code. Perform this changes in CarPawn.cpp (according to the current update in the git): In line 382, change "Val" to "1 – Val". (the complementary value in the range [0.0,1.0]). In line 388, change "Val" to "5Val - 2.5" (Change the range of the given input from [0.0,1.0] to [-1.0,1.0]). In line 404, change "Val" to "4(1 – Val)". (the complementary value in the range [0.0,1.0]).
4. Debug AirSim project (while the steering wheel is connected – it's important).
5. On Unreal Editor, go to Edit->plugins->input devices and enable "Windows RawInput".
6. Go to Edit->Project Settings->Raw Input, and add new device configuration: Vendor ID: 0x046d (In case of Logitech G920, otherwise you might need to check it). Product ID: 0xc261 (In case of Logitech G920, otherwise you might need to check it). Under "Axis Properties", make sure that "GenericUSBController Axis 2", "GenericUSBController Axis 4" and "GenericUSBController Axis 5" are all enabled with an offset of 1.0. Explanation: axis 2 is responsible for steering movement, axis 4 is for brake and axis 5 is for gas. If you need to configure the clutch, it's on axis 3.



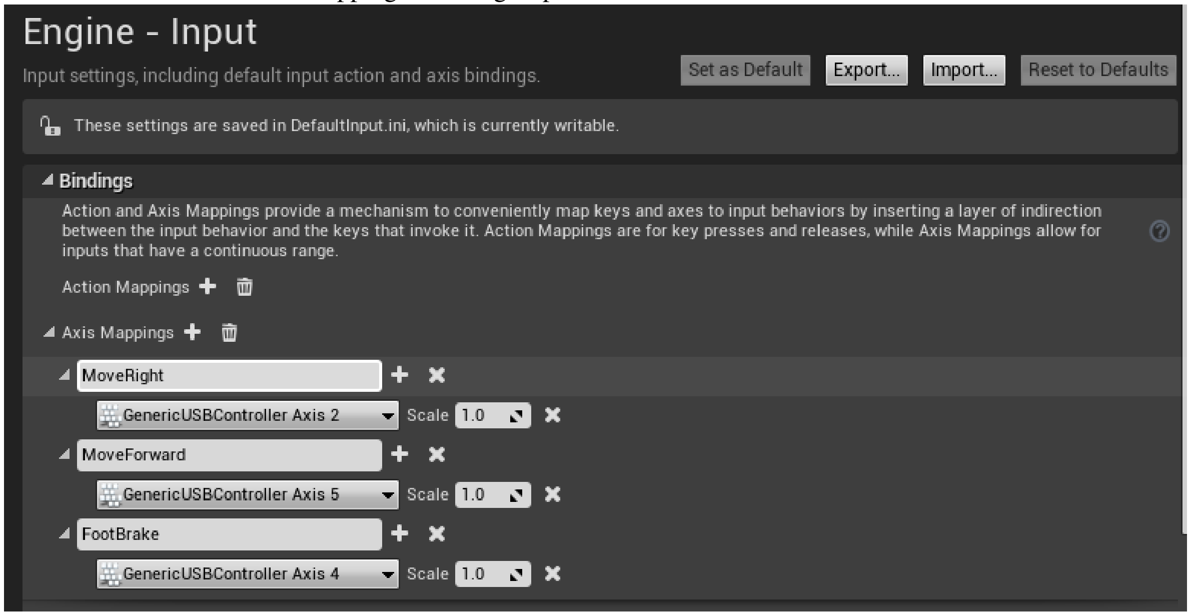
Plugins - Raw Input
Raw Input Settings

These settings are saved in DefaultInput.ini, which is currently writable.

Device Configurations

Device Configurations	1 Array elements
0	4 members
Vendor ID	0x046d
Product ID	0xc261
Axis Properties	8 Array elements
0	4 members
1	4 members
Enabled	✓
Key	GenericUSBController Axis 2
Inverted	
Offset	1.0
2	4 members
3	4 members
4	4 members
5	4 members
6	4 members
7	4 members

7. Go to Edit->Project Settings->Input, Under Bindings in "Axis Mappings": Remove existing mappings from the groups "MoveRight" and "MoveForward". Add new axis mapping to the group "MoveRight", use GenericUSBController axis 2 with a scale of 1.0. Add new axis mapping to the group "MoveForward", use GenericUSBController axis 5 with a scale of 1.0. Add a new group of axis mappings, name it "FootBrake" and add new axis mapping to this group, use GenericUSBController axis 4 with a scale of 1.0.



Engine - Input
Input settings, including default input action and axis bindings.

These settings are saved in DefaultInput.ini, which is currently writable.

Bindings

Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behaviors by inserting a layer of indirection between the input behavior and the keys that invoke it. Action Mappings are for key presses and releases, while Axis Mappings allow for inputs that have a continuous range.

Action Mappings

Axis Mappings

MoveRight
GenericUSBController Axis 2 Scale 1.0
MoveForward
GenericUSBController Axis 5 Scale 1.0
FootBrake
GenericUSBController Axis 4 Scale 1.0

8. Play and drive !

Pay Attention

Notice that in the first time we "play" after debug, we need to touch the wheel to "reset" the values.

Tip

In the gaming software, you can configure buttons as keyboard shortcuts, we used it to configure a shortcut to record dataset or to play in full screen.

2.3.44 Setup Blocks Environment for AirSim

Blocks environment is available in repo in folder `Unreal/Environments/Blocks` and is designed to be lightweight in size. That means its very basic but fast.

Here are quick steps to get Blocks environment up and running:

Windows

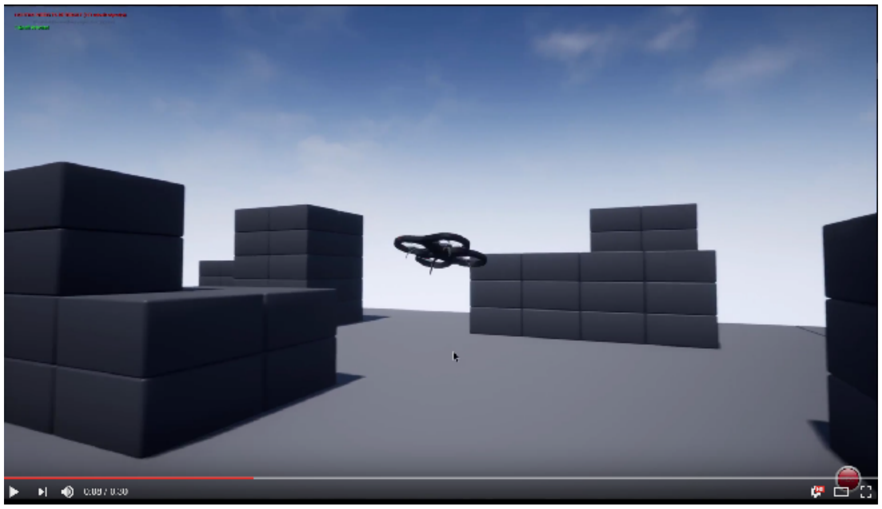
1. Make sure you have installed Unreal and built AirSim.
2. Navigate to folder `AirSim\Unreal\Environments\Blocks` and run `update_from_git.bat`.
3. Double click on generated .sln file to open in Visual Studio 2017 or newer.
4. Make sure `Blocks` project is the startup project, build configuration is set to `DebugGame_Editor` and `Win64`. Hit F5 to run.
5. Press the Play button in Unreal Editor and you will see something like in below video. Also see how to use AirSim.

Changing Code and Rebuilding

For Windows, you can just change the code in Visual Studio, press F5 and re-run. There are few batch files available in folder `AirSim\Unreal\Environments\Blocks` that lets you sync code, clean etc.

Linux

1. Make sure you have built the Unreal Engine and AirSim.
2. Navigate to your UnrealEngine repo folder and run `Engine/Binaries/Linux/UE4Editor` which will start Unreal Editor.
3. On first start you might not see any projects in UE4 editor. Click on Projects tab, Browse button and then navigate to `AirSim/Unreal/Environments/Blocks/Blocks.uproject`.
4. If you get prompted for incompatible version and conversion, select In-place conversion which is usually under "More" options. If you get prompted for missing modules, make sure to select No so you don't exit.
5. Finally, when prompted with building AirSim, select Yes. Now it might take a while so go get some coffee :).
6. Press the Play button in Unreal Editor and you will see something like in below video. Also see how to use AirSim.



Changing Code and Rebuilding

For Linux, make code changes in AirLib or Unreal/Plugins folder and then run `./build.sh` to rebuild. This step also copies the build output to Blocks sample project. You can then follow above steps again to re-run.

Chosing Your Vehicle: Car or Multirotor

By default AirSim spawns multirotor. You can easily change this to car and use all of AirSim goodies. Please see using car guide.

FAQ

These are intermediate files and you can safely ignore it.

2.3.45 Creating and Setting Up Unreal Environment

This page contains the complete instructions start to finish for setting up Unreal environment with AirSim. The Unreal Marketplace has several environment available that you can start using in just few minutes. It is also possible to use environments available on websites such as turbosquid.com or cgitrader.com with bit more effort (here's tutorial video). In addition there also several free environments available.

Below we will use a freely downloadable environment from Unreal Marketplace called Landscape Mountain but the steps are same for any other environments. You can also view these steps performed in Unreal AirSim Setup Video.

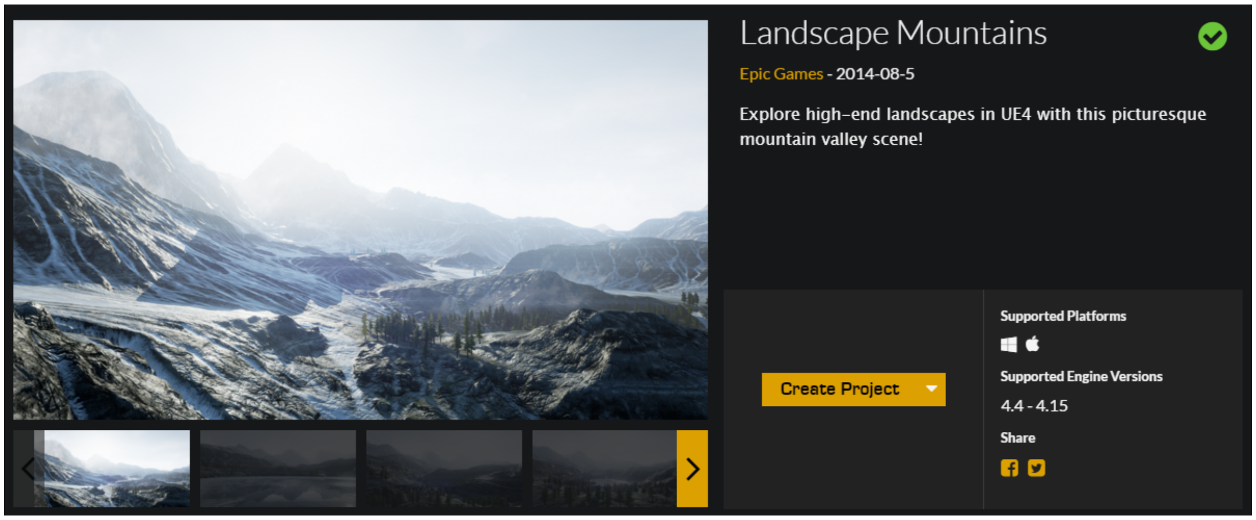
Note for Linux Users

There is no `Epic Games Launcher` for Linux which means that if you need to create custom environment, you will need Windows machine to do that. Once you have Unreal project folder, just copy it over to your Linux machine.

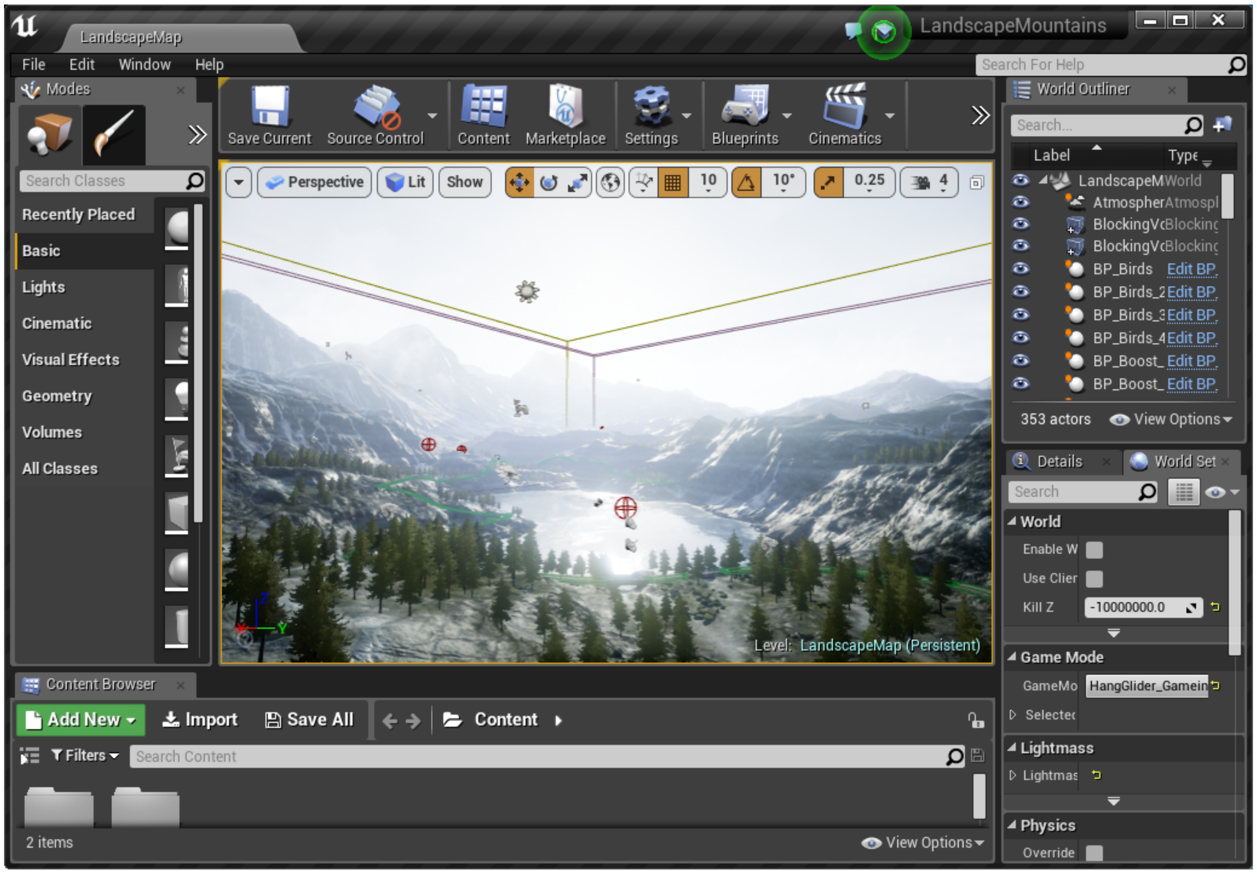
Step by Step Instructions

1. Make sure AirSim is built and Unreal 4.18 is installed as described in build instructions.

2. In `Epic Games Launcher` click the Learn tab then scroll down and find `Landscape Mountains`. Click the `Create Project` and download this content (~2GB download).


Landscape Mountains
Epic Games - 2014-08-5
Explore high-end landscapes in UE4 with this picturesque mountain valley scene!
Create Project
Supported Platforms
Supported Engine Versions
4.4 - 4.15
Share

1. Open `LandscapeMountains.uproject`, it should launch the Unreal Editor.



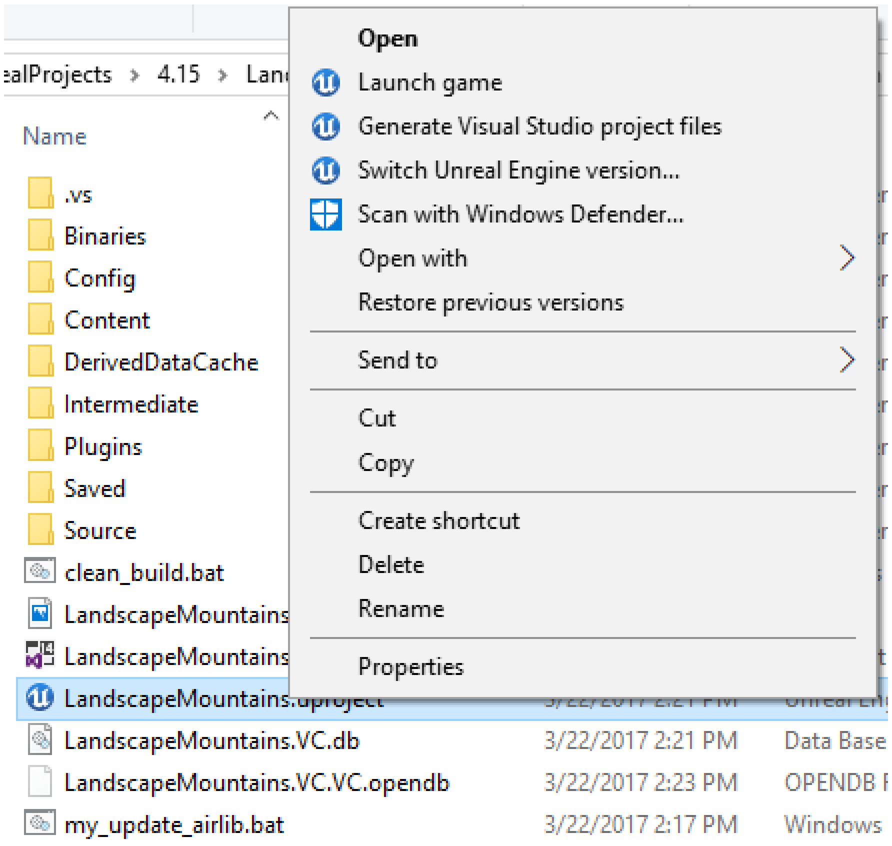
1. From the `File menu` select `New C++ class`, leave default `None` on the type of class, click `Next`, leave default name `MyClass`, and click `Create Class`. We need to do this because Unreal requires at least one source file in project. It should trigger compile and open up Visual Studio solution `LandscapeMountains.sln`.

2. Go to your folder for AirSim repo and copy `Unreal\Plugins` folder in to your `LandscapeMountains`

folder. This way now your own Unreal project has AirSim plugin.

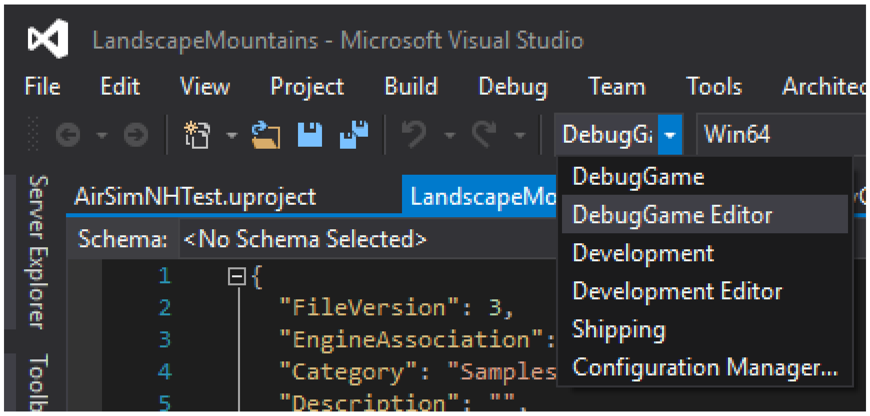
3. Edit the `LandscapeMountains.uproject` so that it looks like this

1. Close Visual Studio and the `Unreal Editor` and right click the LandscapeMountains.uproject in Windows Explorer and select `Generate Visual Studio Project Files`. This step detects all plugins and source files in your Unreal project and generates `.sln` file for Visual Studio.


ealProjects > 4.15 > Lan
Name
.vs
Binaries
Config
Content
DerivedDataCache
Intermediate
Plugins
Saved
Source
clean_build.bat
LandscapeMountains
LandscapeMountains
LandscapeMountains.uproject
LandscapeMountains.VC.db 3/22/2017 2:21 PM Data Base
LandscapeMountains.VC.VC.opendb 3/22/2017 2:23 PM OPENDB F
my_update_airlib.bat 3/22/2017 2:17 PM Windows
Open
Launch game
Generate Visual Studio project files
Switch Unreal Engine version...
Scan with Windows Defender...
Open with
Restore previous versions
Send to
Cut
Copy
Create shortcut
Delete
Rename
Properties

Tip: If the `Generate Visual Studio Project Files` option is missing you may need to reboot your machine for the Unreal Shell extensions to take effect. If it is still missing then open the LandscapeMountains.uproject in the Unreal Editor and select `Refresh Visual Studio Project` from the `File` menu.

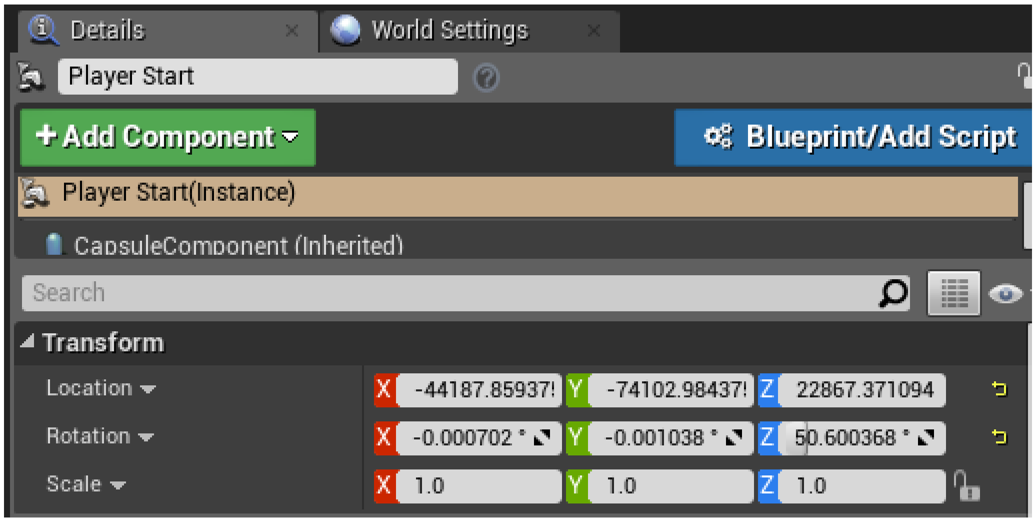
1. Reopen `LandscapeMountains.sln` in Visual Studio, and make sure "DebugGame Editor" and "Win64" build configuration is the active build configuration.



```
{
    "FileVersion": 3,
    "EngineAssociation":
    "Category": "Samples
    "Description": "",
```

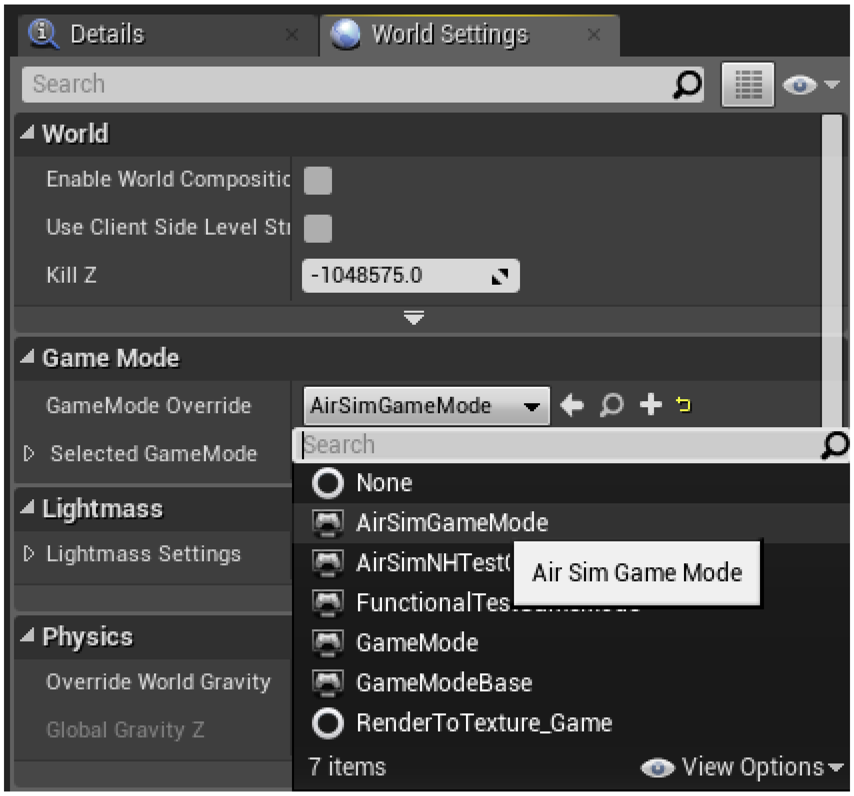
1. Press `F5` to `run`. This will start the Unreal Editor. The Unreal Editor allows you to edit the environment,

assets and other game related settings. First thing you want to do in your environment is set up `PlayerStart` object. In Landscape Mountains environment, `PlayerStart` object already exist and you can find it in the `World Outliner`. Make sure its location is setup as shown. This is where AirSim plugin will create and place the vehicle. If its too high up then vehicle will fall down as soon as you press play giving potentially random behavior



	X	Y	Z
Location	-44187.85937!	-74102.98437!	22867.371094
Rotation	-0.000702 °	-0.001038 °	50.600368 °
Scale	1.0	1.0	1.0

1. In `Window/World Settings` as shown below, set the `GameMode Override` to `AirSimGameMode`:



1. Go to 'Edit->Editor Preferences' in Unreal Editor, in the 'Search' box type 'CPU' and ensure that the 'Use Less CPU when in Background' is unchecked. If you don't do this then UE will be slowed down dramatically when UE window loses focus.
2. Be sure to `Save` these edits. Hit the Play button in the Unreal Editor. See how to use AirSim.

Congratulations! You are now running AirSim in your own Unreal environment.

Choosing Your Vehicle: Car or Multirotor

By default AirSim prompts user for which vehicle to use. You can easily change this by setting SimMode. Please see using car guide.

Updating Your Environment to Latest Version of AirSim

Once you have your environment using above instructions, you should frequently update your local AirSim code to latest version from GitHub. Below are the instructions to do this:

1. First put clean.bat (or clean.sh for Linux users) in the root folder of your environment. Run this file to clean up all intermediate files in your Unreal project.
2. Do `git pull` in your AirSim repo followed by `build.cmd` (or `./build.sh` for Linux users).
3. Replace [your project]/Plugins folder with AirSim/Unreal/Plugins folder.
4. Right click on your .uproject file and chose "Generate Visual Studio project files" option. This is not required for Linux.

FAQ

What are other cool environments?

Unreal Marketplace has dozens of prebuilt extra-ordinarily detailed environments ranging from Moon to Mars and everything in between. The one we have used for testing is called Modular Neighborhood Pack but you can use any environment. Another free environment is Infinity Blade series. Alternatively, if you look under the Learn tab in Epic Game Launcher, you will find many free samples that you can use. One of our favorites is "A Boy and His Kite" which is a 100 square miles of highly detailed environment (caution: you will need *very* beefy PC to run it!).

When I press Play button some kind of video starts instead of my vehicle.

If the environment comes with MatineeActor, delete it to avoid any startup demo sequences. There might be other ways to remove it as well, for example, click on Blueprints button, then Level Blueprint and then look at Begin Play event in Event Graph. You might want to disconnect any connections that may be starting "matinee".

Is there easy way to sync code in my Unreal project with code in AirSim repo?

Sure, there is! You can find bunch of `.bat` files (for linux, `.sh`) in `AirSim\Unreal\Environments\Blocks`. Just copy them over to your own Unreal project. Most of these are quite simple and self explanatory.

I get some error about map.

You might have to set default map for your project. For example, if you are using Modular Neighborhood Pack, set the Editor Starter Map as well as Game Default Map to Demo_Map in Project Settings > Maps & Modes.

I see "Add to project" option for environment but not "Create project" option.

In this case, create a new blank C++ project with no Starter Content and add your environment in to it.

I already have my own Unreal project. How do I use AirSim with it?

Copy the `Unreal\Plugins` folder from the build you did in the above section into the root of your Unreal project's folder. In your Unreal project's .uproject file, add the key `AdditionalDependencies` to the "Modules" object as we showed in the `LandscapeMountains.uproject` above.

and the `Plugins` section to the top level object:

2.3.46 Unreal Environment

Setting Up the Unreal Project

Option 1: Built-in Blocks Environment

To get up and running fast, you can use the Blocks project that already comes with AirSim. This is not very highly detailed environment to keep the repo size reasonable but we use it for various testing all the times and it is the easiest way to get your feet wet in this strange land.

Follow these quick steps.

Option 2: Create Your Own Unreal Environment

If you want to setup photo-realistic high quality environments, then you will need to create your own Unreal project. This is little bit more involved but worthwhile!

Follow this step-by-step guide.

Changing Code and Development Workflow

To see how you can change and test AirSim code, please read our recommended development workflow.

2.3.47 Upgrading to Unreal Engine 4.18

These instructions apply if you are already using AirSim on Unreal Engine 4.16. If you have never installed AirSim, please see How to get it.

Caution: The below steps will delete any of your unsaved work in AirSim or Unreal folder.

Do this first

For Windows Users

1. Install Visual Studio 2017 with VC++, Python and C#.
2. Install UE 4.18 through Epic Games Launcher.
3. Start `x64 Native Tools Command Prompt for VS 2017` and navigate to AirSim repo.

4. Run `clean_rebuild.bat` to remove all unchecked/extra stuff and rebuild everything.

For Linux Users

1. From your AirSim repo folder, run 'clean_rebuild.sh'.
2. Rename or delete your existing folder for Unreal Engine.
3. Follow step 1 and 2 to install Unreal Engine 4.18.

Upgrading Your Custom Unreal Project

If you have your own Unreal project created in an older version of Unreal Engine then you need to upgrade your project to Unreal 4.18. To do this,

1. Open .uproject file and look for the line `"EngineAssociation"` and make sure it reads like `"EngineAssociation": "4.18"`.
2. Delete `Plugins/AirSim` folder in your Unreal project's folder.
3. Go to your AirSim repo folder and copy `Unreal\Plugins` folder to your Unreal project's folder.
4. Copy *.bat (or *.sh for Linux) from `Unreal\Environments\Blocks` to your project's folder.
5. Run `clean.bat` (or `clean.sh` for Linux) followed by `GenerateProjectFiles.bat` (only for Windows).

FAQ

I have an Unreal project that is older than 4.16. How do I upgrade it?

Option 1: Just Recreate Project

If your project doesn't have any code or assets other than environment you downloaded then you can also simply recreate the project in Unreal 4.18 Editor and then copy Plugins folder from `AirSim/Unreal/Plugins`.

Option 2: Modify Few Files

Unreal versions newer than Unreal 4.15 has breaking changes. So you need to modify your *.Build.cs and *.Target.cs which you can find in the `Source` folder of your Unreal project. So what are those changes? Below is the gist of it but you should really refer to Unreal's official 4.16 transition post.

In your project's *.Target.cs

1. Change the contructor from, `public MyProjectTarget(TargetInfo Target)` to `public MyProjectTarget(TargetInfo Target) : base(Target)`
2. Remove `SetupBinaries` method if you have one and instead add following line in contructor above: `ExtraModuleNames.AddRange(new string[] { "MyProject" });`

In your project's *.Build.cs

Change the constructor from `public MyProject(TargetInfo Target)` to `public MyProject(ReadOnlyTargetRules Target) : base(Target)`.

And finally…

Follow above steps to continue the upgrade. The warning box might show only "Open Copy" button. Don't click that. Instead, click on More Options which will reveal more buttons. Choose `Convert-In-Place option`. *Caution:* Always keep backup of your project first! If you don't have anything nasty, in place conversion should go through and you are now on the new version of Unreal.

2.3.48 Upgrading API Client Code

There have been several API changes in AirSim v1.2 that we hope removes inconsistency, adds future extensibility and presents cleaner interface. Many of these changes are however *breaking changes* which means you will need to modify your client code that talks to AirSim.

Quicker Way

While most changes you need to do in your client code are fairly easy, a quicker way is simply to take a look at the example code such as Hello Droneor Hello Car to get gist of changes.

Importing AirSim

Instead of,

```
from AirSimClient import *
```

use this:

```
import airsim
```

Above assumes you have installed AirSim module using,

If you are running you code from PythonClient folder in repo then you can also do this:

```
import setup_path
import airsim
```

Here setup_path.py should exist in your folder and it will set the path of `airsim` package in `PythonClient` repo folder. All examples in PythonClient folder uses this method.

Using AirSim Classes

As we have everything now in package, you will need to use explicit namespace for AirSim classes like shown below.

Instead of,

```
client1 = CarClient()
```

use this:

```
client1 = airsim.CarClient()
```

AirSim Types

We have moved all types in `airsim` namespace.

Instead of,

```
image_type = AirSimImageType.DepthVis

d = DrivetrainType.MaxDegreeOfFreedom
```

use this:

```
image_type = airsim.ImageType.DepthVis

d = airsim.DrivetrainType.MaxDegreeOfFreedom
```

Getting Images

Nothing new below, it's just combination of above. Note that all APIs that previously took `camera_id`, now takes `camera_name` instead. You can take a look at available cameras here.

Instead of,

```
responses = client.simGetImages([ImageRequest(0, AirSimImageType.DepthVis)])
```

use this:

```
responses = client.simGetImages([airsim.ImageRequest("0", airsim.ImageType.DepthVis)])
```

Utility Methods

In earlier version, we provided several utility methods as part of `AirSimClientBase`. These methods are now moved to `airsim` namespace for more pythonic interface.

Instead of,

```
AirSimClientBase.write_png(my_path, img_rgba)

AirSimClientBase.wait_key('Press any key')
```

use this:

```
airsim.write_png(my_path, img_rgba)

airsim.wait_key('Press any key')
```

Camera Names

AirSim now uses names to reference cameras instead of index numbers. However to retain backward compatibility, these names are aliased with old index numbers as string.

Instead of,

```
client.simGetCameraInfo(0)
```

use this:

```
client.simGetCameraInfo("0")

# or

client.simGetCameraInfo("front-center")
```

Async Methods

For multirotors, AirSim had various methods such as `takeoff` or `moveByVelocityZ` that would take long time to complete. All of such methods are now renamed by adding the suffix *Async* as shown below.

Instead of,

```
client.takeoff()

client.moveToPosition(-10, 10, -10, 5)
```

use this:

```
client.takeoffAsync().join()

client.moveToPositionAsync(-10, 10, -10, 5).join()
```

Here `.join()` is a call on Python's `Future` class to wait for the async call to complete. You can also choose to do some other computation instead while the call is in progress.

Simulation-Only Methods

Now we have clear distinction between methods that are only available in simulation from the ones that may be available on actual vehicle. The simulation only methods are prefixed with `sim` as shown below.

State Information

Previously `CarState` mixed simulation-only information like `kinematics_true`. Moving forward, `CarState` will only contain information that can be obtained in real world.

```
k = car_state.kinematics_true
```

use this:

```
k = car_state.kinematics_estimated

# or

k = client.simGetGroundTruthKinematics()
```

2.3.49 Upgrading Settings

The settings schema in AirSim 1.2 is changed for more flexibility and cleaner interface. If you have older settings.json file then you can either delete it and restart AirSim or use this guide to make manual upgrade.

Quicker Way

We recommend simply deleting the settings.json and add back the settings you need. Please see the doc for complete information on available settings.

Changes

UsageScenario

Previously we used `UsageScenario` to specify the `ComputerVision` mode. Now we use `"SimMode": "ComputerVision"` instead.

CameraDefaults and Changing Camera Settings

Previously we had `CaptureSettings` and `NoiseSettings` in root. Now these are combined in new `CameraDefaults` element. The schema for this element is later used to configure cameras on vehicle.

Gimbal

The Gimbal element (instead of old Gimble element) is now moved out of `CaptureSettings`.

CameraID to CameraName

All settings now reference cameras by name instead of ID.

Using PX4

The new Vehicles element allows to specify which vehicles to create. To use PX4, please see this section.

AdditionalCameras

The old `AdditionalCameras` setting is now replaced by Cameras element within vehicle setting.

2.3.50 How to Use Car in AirSim

By default AirSim prompts user for which vehicle to use. You can easily change this by setting SimMode. For example, if you want to use car instead then just set the SimMode in your settings.json which you can find in your `~/Documents/AirSim` folder, like this:

Now when you restart AirSim, you should see the car spawned automatically.

Manual Driving

Please use the keyboard arrow keys to drive manually. Spacebar for the handbrake. In manual drive mode, gears are set in "auto".

Using APIs

You can control the car, get state and images by calling APIs in variety of client languages including C++ and Python. Please see APIs doc for more details.

Changing Views

By default camera will chase the car from the back. You can get the FPV view by pressing `F` key and switch back to chasing from back view by pressing `/` key. More keyboard shortcuts can be seen by pressing F1.

Cameras

By default car is installed with 5 cameras: center, left and right, driver and reverse. You can chose the images from these camera by specifying the name.

2.3.51 What's new

Below is highly summerized curated list of important changes. This does not include minor/less important changes or bug fixes or things like documentation update. This list updated every few months. For full list of changes, please review commit history.

June, 2018

- Development workflow doc
- Better Python 2 compatibility
- OSX setup fixes
- Almost complete rewrite of our APIs with new threading model, merging old APIs and creating few newer ones

April, 2018

- Upgraded to Unreal Engine 4.18 and Visual Studio 2017
- API framework refactoring to support world-level APIs
- Latest PX4 firmware now supported
- CarState with more information
- ThrustMaster wheel support
- pause and continueForTime APIs for drone as well as car
- Allow drone simulation run at higher clock rate without any degradation
- Forward-only mode fully functional for drone (do orbits while looking at center)
- Better PID tuning to reduce wobble for drones

- Ability to set arbitrary vehicle blueprint for drone as well as car
- gimbal stabilization via settings
- Ability to segment skinned and skeletal meshes by their name
- moveByAngleThrottle API
- Car physics tuning for better maneuverability
- Configure additional cameras via settings
- Time of day with geographically computed sun position
- Better car steering via keyboard
- Added MeshNamingMethod in segmentation setting
- gimbal API
- getCameraParameters API
- Ability turn off main rendering to save GPU resources
- Projection mode for capture settings
- getRCData, setRCData APIs
- Ability to turn off segmentation using negative IDs
- OSX build improvements
- Segmentation working for very large environments with initial IDs
- Better and extensible hash calculation for segmentation IDs
- Extensible PID controller for custom integration methods
- Sensor architecture now enables renderer specific features like ray casting
- Laser altimeter sensor

Jan 2018

- Config system rewrite, enable flexible config we are targeting in future
- Multi-Vehicle support Phase 1, core infrastructure changes
- MacOS support
- Infrared view
- 5 types of noise and interference for cameras
- WYSIWIG capture settings for cameras, preview recording settings in main view
- Azure support Phase 1, enable configurability of instances for headless mode
- Full kinematics APIs, ability to get pose, linear and angular velocities + accelerations via APIs
- Record multiple images from multiple cameras
- New segmentation APIs, ability to set configure object IDs, search via regex
- New object pose APIs, ability to get pose of objects (like animals) in environment
- Camera infrastructure enhancements, ability to add new image types like IR with just few lines
- Clock speed APIs for drone as well as car, simulation can be run with speed factor of 0 < x < infinity

- Support for Logitech G920 wheel
- Physics tuning of the car, Car doesn't roll over, responds to steering with better curve, releasing gas paddle behavior more realistic
- Debugging APIs
- Stress tested to 24+ hours of continuous runs
- Support for Landscape and sky segmentation
- Manual navigation with accelerated controls in CV mode, user can explore environment much more easily
- Collison APIs
- Recording enhancements, log several new data points including ground truth, multiple images, controls state
- Planner and Perspective Depth views
- Disparity view
- New Image APIs supports float, png or numpy formats
- 6 config settings for image capture, ability to set auto-exposure, motion blur, gamma etc
- Full multi-camera support through out including sub-windows, recording, APIs etc
- Command line script to build all environments in one shot
- Remove submodules, use rpclib as download

Nov 2017

- We now have the car model.
- No need to build the code. Just download binaries and you are good to go!
- The reinforcement learning example with AirSim
- New built-in flight controller called simple_flight that "just works" without any additional setup. It is also now *default*.
- AirSim now also generates depth as well as disparity images that is in camera plan.
- We also have official Linux build now! If you have been using AirSim with PX4, you might want to read the release notes.

2.3.52 Who is Using AirSim?

Would you like to see your own group or project here?

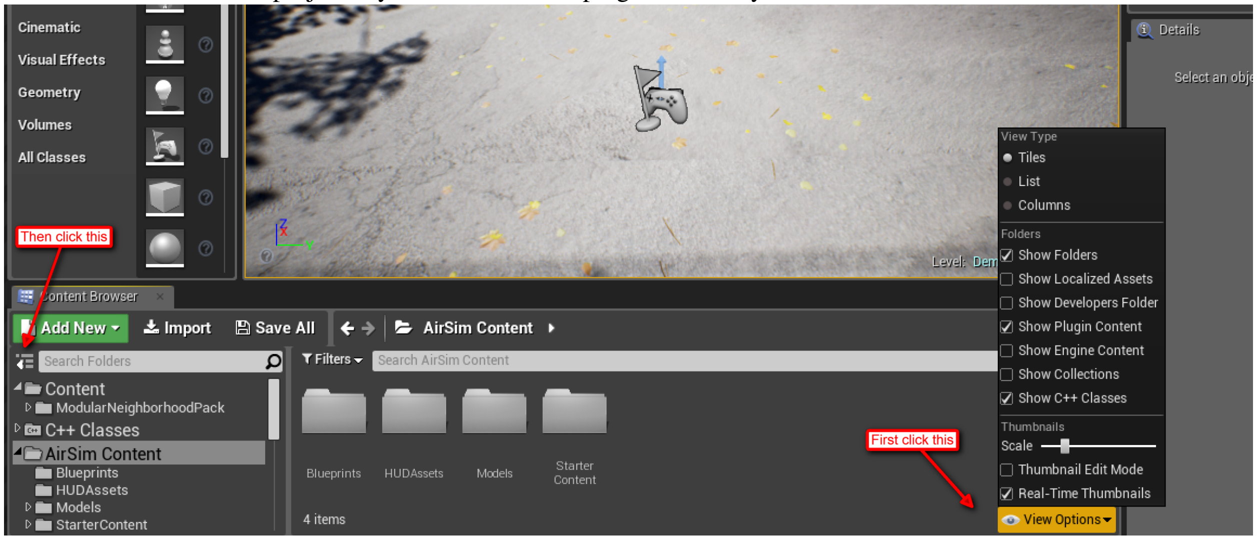
Just add a GitHub issue with quick details and link to your website or paper or send us email at msrair at microsoft.com.

- NASA Ames Research Center – Systems Analysis Office
- Astrobotic
- GRASP Lab, Univ of Pennsylvania
- Department of Aeronautics and Astronautics, Stanford University
- Formula Technion
- Ghent University

- ICARUS
- UC, Santa Barbara
- WISE Lab, Univ of Waterloo
- HAMS project, MSR India
- Washington and Lee University
- University of Oklahoma
- Robotics Institute, Carnegie Mellon University
- Texas A&M
- Robotics and Perception Group, University of Zurich
- National University of Ireland, Galway (NUIG)

2.3.53 How to use plugin contents

Plugin contents are not shown in Unreal projects by default. To view plugin content, you need to click on few semi-hidden buttons:



Causion Changes you make in content folder are changes to binary files so be careful.