# Mississippi State University Autonomous Vehicle Simulator Python Interface (MAVS-Python)

User Guide

Center for Advanced Vehicular Systems
Mississippi State University

# Contents

# Chapter 1

# Introduction

This User Guide documents the MAVS-Python interface. MAVS is written in C++ and features interacting simulations of the vehicle and sensors that constitute an autonomous ground vehicle (AGV), as well as the environment and terrain of the AGV simulation.

MAVS has been used for a variety of applications including studying lidar interaction with vegetation [3], calculating the best placement for sensors on a vehicle [4][6], training vision-based classification algorithms [1], and studying lidar interaction with rain [2].

More information about MAVS can be found on the Wiki. Usage of the C++ API can be found on the API documentation site.

## 1.1 Installation

If you have access to the repository, MAVS can be built from source. Otherwise, the MAVS-Python package can be used with the pre-built Windows 10 dll.

To use the binary package, copy the "mavs_windows10" folder somewhere on your local file system. Open a terminal in the folder and run the installation script:

```
python install_mavs_windows10.py
```

MAVS will be installed in the "AppData/Local/mavs" folder. Next, add the MAVS dll to the system path following the instructions in the attached PowerPoint file. You may need to restart your computer after adding MAVS to the system path.

## 1.2 Including MAVS-Python

To use the MAVS dll in your python code, you must have python load it to the system path.

```
import sys

# Set the path to the mavs python api, mavs_interface.py
# You will have to change the file path on your system.
sys.path.append(r'C:\Users\cgoodin\Desktop\vm_shared\
                shared_repos\mavs\src\mavs_python')

# Load the mavs python modules
import mavs_interface as mavs
import mavs_python_paths
```

## 1.3 The MAVS Coordinate System

The default MAVS coordinate system is known as East-North-Up (ENU). The origin of the local ENU coordinate system is always (0,0,0). The origin can correspond to a real-world latitude, longitude, and elevation. By default, the origin of a MAVS scene is $32.033^o$ Northing, $90.8742^o$ Westing, with an elevation of 73.152 meters above sea level. Note that in MAVS, Westing is positive longitude while Easting is negative. The location of the origin can be defined in the simulation input file for a MAVS simulation (Chapter 2).

The default orientation is MAVS is with the positive $x$-axis east, the positive $y$-axis as north, and the positive $z$-axis pointing up. For individual objects, their default orientation is with positive $x$ facing forward and positive $z$ pointing up, see Figure 1.1. The length scale in MAVS is meters.

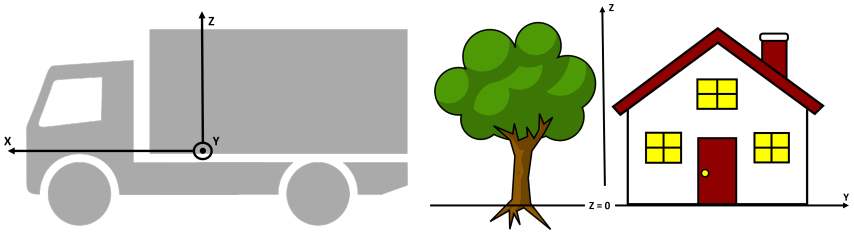


Figure 1.1: MAVS coordinate system

# Chapter 2

# MAVS-Python Simulation

A MAVS-Python simulation consists of a scene, an environment, a vehicle, a vehicle controller, sensors, and a waypoint list for the vehicle to follow.

## 2.1 Creating and Running a MAVS Simulation

A simulation can be created through the Mavs-Python interface.

```
simulation = mavs.MavsSimulation()
```

The simulation must be initialized with an input file

```
sim_file = 'sims/sensor_sims/halo_city_sim.json'
simulation.Load(mavs_python_paths.mavs_data_path +
                '/' + sim_file)
```

The simulation is run in a python loop.

```
dt = 1.0/30.0 # 30 Hz update rate
while True:
    simulation.Update(dt, throttle=1.0,
                      steering=0.0,
                      braking=0.0,
                      update_actor=True)
```

The throttle, steering, and braking command may come from a waypoint follower or keyboard input (Chapter 7). The "update_actor" variable species whether or not to update the position of the mesh actor for the vehicle (Chapter 5).

## 2.2 The Simulation Input File

In the previous section, the input file "halo_city_sim.json" was loaded. The file has sections for

- **Environment** - The context of the simulation. Includes environmental conditions like rain and fog as well as pointer to a specific scene.
- **Scene** - The geometry of the static and dynamic objects in the digital scene.
- **Poses** - The path that the vehicle will follow, if not driven manually.
- **Sensors** - A list of the sensors on the vehicle and their properties.
- **Vehicle** - The vehicle input file.
- **Controller** - Vehicle controller parameters for waypoint following. Must be matched to the vehicle.

When a simulation is created from an input file, the vehicle, sensors, actors, environment, and scene are all loaded automatically. However, these simulation components can be modified programatically through the MAVS-Python API as well. Each of these items, including the parameters and how to access them via the MAVS-Python interface, will be discussed in the following sections. Example simulation inputs are found in the Mavs-Python data directory under "data/sims/sensor_sims".

# Chapter 3

# MAVS-Python Environment

The MAVS-Python Environment has information about the geometry of the scene, the dynamic objects and actors moving in that scene, and the environmental conditions and context of the simulation.

A new environment can be created.

```
env = mavs.MavsEnvironment()
```

Note that this creates a pointer to a MAVS class that must be freed if the environment is going to be reassigned.

```
env.DeleteEnvironment()
env = mav_interface.MavsEnvironment()
```

Environmental conditions can be edited with the following functions

```
# Rain rate in mm/h
env.SetRainRate(10.0)

# Haziness factor, 2-10
env.SetTurbidity(4.0)

# Local albedo, 0-1.0
env.SetAlbedo(0.05)

# Fog density, 0-100
env.SetFog(10.0)

# Cloud-cover fraction, 0-1.0
env.SetCloudCover(0.45)
```

```
# Snowfall rate, mm/h
env.SetSnow(10.0)

# Wind speed (m/s), east and north directions
env.SetWind([2.5, 1.5])
```

A scene (defining the objects / geometry) must be created separately and added to the environment, which will be discussed in the following chapter.

To update the dynamic objects in the scene, including the actors and animations (Chapter 5), particle systems like dust and smoke, and falling snow, the environment simulation time must be advanced. In order to advance the state of the simulation time by $dt$ seconds:

```
dt = 0.1
env.AdvanceTime(dt)
```

# Chapter 4

# MAVS-Python Scene

MAVS scenes are comprised of meshes. The default MAVS mesh format is Wavefront .obj.

The MAVS-Python base class for scenes is called "MavsScene()". Scenes can either be loaded from a scene definition file (MavsEmbreeScene()), or a random scene can be created using input parameters (MavsRandomScene()). Both MavsEmbreeScene() and MavsRandomScene() inherit from the base MavsScene() class.

## 4.1 Loading a MAVS-Python Scene From File

Example scene files are in the MAVS data directory under the "scenes" folder. To create a scene from an existing scene file:

```
scene = mavs.MavsEmbreeScene()
scenefile = 'scenes/cube_scene.json'
scene.Load(mavs_python_paths.mavs_data_path + '/' +
           scenefile)
```

## 4.2 Creating a Random MAVS-Python Scene

To create a random scene, first declare a random scene object.

```
scene = mavs.MavsRandomScene()
```

Next, define aspects of the terrain geometry like the size and roughness.

```
# terrain width and length in meters
```

```
scene.terrain_width = 100.0
scene.terrain_length = 100.0
# magnitude of low frequency roughness
# in meters (rolling hills), 0=flat
scene.lo_mag = 2.0
# magnitude of hi-freq roughness in meters
# (bumpy terrain), 0 = smooth
scene.hi_mag = 0.1
# plant density should be from [0-1].
# 0=no plants, 1=most plants
scene.plant_density = 0.05
```

To include potholes in the scene, the average depth and diameter of the holes must be specified, as well a specification of each pothole position.

```
scene.pothole_depth = 0.5
scene.pothole_diameter = 1.0
scene.AddPotholeAt(10.0, -12.0)
```

To create trails, the width of the trail, the width of the track marks, and the width between the track-marks in the trail must be specified. Trails are automatically created to follow the terrain. The options are 'Ridges', 'Loop', or 'Valleys'

```
# trail properties
scene.trail_width = 2.0
scene.track_width = 0.3
scene.wheelbase = 1.8
scene.path_type = 'Ridges'
```

The random scene ecosystem defines the type and amount of plants present in the scene. Example ecosystem files are in the MAVS data folder under the "ecosystem_files" directory.

```
scene.eco_file = 'american_pine_forest.json'
```

Finally, give the scene a name (no extension) and create the scene with the defined parameters.

```
scene.basename = 'pothole_scene'
scene.CreateScene()
```

## 4.3 Adding the scene to the environment

A MavsScene() contains a scene object which is a pointer to the MAVS data structure, in addition to other class data. To directly access the scene pointer, call the scene data member. The scene member must be added to an environment in order for the environment to be used in sensor and vehicle simulations.

```
env.SetScene(scene.scene)
```

## 4.4 Other scene functions

Scene labeling must be turned on in order to generate labeled training data with lidar and camera sensors. By default, scene labeling is turned off. To turn it on:

```
scene.TurnOnLabeling()
```

Scene labeling comes with some computational overhead which may slow the simulation down slightly. To turn it back off:

```
scene.TurnOffLabeling()
```

# Chapter 5

# MAVS-Python Actors and Animations

A MAVS-Python environment may include dynamic moving objects in the form of either actors or animations.

Actors are dynamic objects that move through the scene (without animation), with physics typically controlled externally from the MAVS core library. An example might include a thrown ball or a UAV flying through the scene.

Animations are dynamic objects that include a sequence of keyframes describing the motion of the animation. A typical example is a walking pedestrian.

## 5.1 Adding animations to a MAVS-Python Scene

An animation is any object that moves. Example animation definition files can be found in "data/scenes/meshes/animations" folder. To add an animation to the scene via animation file, first create the animation.

```
animation = mavs.MavsAnimation()
anim_folder = "/scenes/meshes/animations/GenericWalk"
animation.Load((mavs_data_path + anim_folder),
(mavs_data_path+anim_folder+"/walk_frames.txt"))
```

The first argument is the path to the folder containing the frames of the animation. The second argument is a file that lists the sequence of those frames.

Next, properties of the animation meshes can be set such as the mesh scale factor

```
animation.SetScale(0.01)
# Rotate y to x, Rotate y to z?
animation.SetRotations(True,False)
animation.SetBehavior('straight')
```

The behavior options for an animation are "wander", "straight", "circle", and "crowd". Next, the initial position, heading, and speed of the animation can be set.

```
animation.SetPosition(0.0, 0.0)
animation.SetHeading(-math.pi/2.0)
animation.SetSpeed(5.0)
```

Finally, the animation can be added to the scene.

```
scene.AddAnimation(animation)
```

The animation will be moved automatically when the environment update function is called.

```
dt = 1.0/30.0
env.AdvanceTime(dt)
```

## 5.2 MAVS-Python Actors

To add an actor to a MAVS-Python simulation, first load the actor input file to the environment.

```
actorfile = "actors/forester_actor.json"
actornum = env.AddActor(mavs_python_paths.mavs_data_path +
                        '/' + actorfile)
```

Where "actornum" is an integer identifier for the actor that was just added. Example actor files can be found in the MAVS data folder under the "actors/actors" directory.

Moving actors can generate dust. To add dust to an actor

```
env.AddDustToActor(actornum)
```

Like an animation, an actor may be automatically updated when environment time advances, or its position may be set explicitly at each time step if there is some external process controlling its physics.

```
new_position = [1.0, 0.0, 2.0]
new_heading = 0.2
new_orientation = [math.cos(0.5*new_heading), 0.0, 0.0,
                   math.sin(0.5*new_heading)]
env.SetActorPosition(actornum, new_heading, new_orientation)
```

# Chapter 6

# MAVS Sensor Simulations

There are four basic sensor types in MAVS-Python: camera, lidar, radar, and RTK. The initialization, implementation, and update of these sensors through the MAVS-Python interface is discussed in the following sections. The MavsSensor() base class defines many of the functions these sensors have in common.

In MAVS-Python, sensors mounted to a vehicle can specify an offset from the vehicle center-of-gravity.

```
relative_offset = [1.0, 0.0, 0.6]
relative_orientation = [1.0, 0.0, 0.0, 0.0]
sensor.SetOffset(relative_offset, relative_orientation)
```

where "sensor" is any MAVS-Python sensor.

Before a sensor is updated, its position must be set. Note that the position and orientation are of the vehicle the sensor is attached to. The offset from the previous command is applied automatically.

```
current_position = [52.0, 17.3, 2.56]
current_orientation = [0.7071, 0.0, 0.0, 0.7071]
sensor.SetPose(current_position, current_orientation)
```

Once the position is set, the sensor calculations are performed with a call to "Update"

```
dt = 1.0/30.0
sensor.Update(env,dt)
```

where "env" is a MavsEnvironment (Chapter 3). This will generate a new image, point cloud, radar return, or odometry reading based on the current

sensor and environment state. The update function must be called at the same frequency that the sensor is updated.

Other common sensor functions include annotating a frame and saving labeled data or saving raw, unlabeled data.

```
sensor.AnnotateFrame(env)
sensor.SaveAnnotation(env,'annotation_file.txt')
sensor.SaveRaw() # named automatically
```

Sensors like lidar and cameras also implement a display method which opens a window to display the current output to the screen.

```
sensor.Display()
```

Each type of sensor also has functions unique to that sensor which will be discussed in the following sections.

## 6.1 LIDAR

A MAVS lidar is created with a type using the following command.

```
type = 'HDL-32E'
lidar = mavs.MavsLidar(type)
```

The available types are "M8", "HDL-64E", "HDL-32E", "VLP-16", "LMS-291", "OS1", "OS2", and "RS32".

The default sensor display command (sensor.Display()) show the lidar point cloud from a top-down view. To show a view from the perspective of the lidar:

```
lidar.DisplayPerspective()
```

Point clouds can be saved in column text files with columns for x-y-z in the sensor frame and additional columns for color

```
fname = 'colorized_cloud.pts'
lidar.SaveColorizedPointCloud(fname)
```

or by point label

```
fname = 'labeled_cloud.pts'
lidar.SaveLabeledPointCloud(fname)
```

Additionally, a python list containing the points (registered to world coordinates) can be accessed with

```
points = lidar.GetPoints()
```

This will return a python array of dimension [$n_p$,3] where $n_p$ is the number of points and the columns are x-y-z.

Labeled points in the sensor frame can be accessed by

```
labeled_points = lidar.GetUnRegisteredPointsXYZIL()
```

This will return a python array of dimension [$n_p$,5] where the columns are x-y-z-intensity-label.

The point cloud can alsoe be saved in the Point Cloud Library PCD format.

```
lidar.SavePcd("pcd_file.pcd")
# or to save labeled pcd file
lidar.SaveLabeledPcd("labeled_pcd_file.pcd")
```

## 6.2 Camera

MAVS cameras can either be created using a predefined model or with user-specified parameters.

```
camera = mavs.MavsCamera()
```

To load an existing camera

```
camera_type = 'XCD-V60'
camera.Model(camera_type)
```

Available camera types are "XCD-V60", "Flea", "HD1080", and "MachineVision". Otherwise, to create a custom camera

```
u_pix = 256
v_pix = 256
u_res = 0.035 # meters
v_res = 0.035 # meters
flen = 0.035 # meters
camera.Initialize(u_pix, v_pix, u_res, v_res, flen)
```

Before calling the update function, the environmental conditions for the camera should be set if the environment has changed since the last time step. This is a precalculation step for the lighting model that is specific to the camera.

```
camera.SetEnvironmentProperties(env)
```

After updating the camera, to save the current image to a file

```
fname = 'current_image.bmp'
camera.SaveCameraImage(fname)
```

Other aspects of the rendering and camera properties can be set through the interface.

```
# Not rendering shadows speeds up the calculation
camera.RenderShadows(False)
# Increasing the anti-aliasing gives more realism,
# but slows down the simulation
camera.SetAntiAliasingFactor(1)
# If raining, add raindrop splatter on the lens
camera.SetDropsOnLens(True)
# Set the digital compression and gain of the camera
camera.SetGammaAndGain(0.5,0.95)
```

Finally, the camera sensor can also be used in MAVS-Python to interact with the simulation. While sensors are typically either stationary or attached to a moving vehicle in MAVS, the camera can be "freed" so that its pose is controlled with the keyboard. To free a camera:

```
camera.FreePose()
```

When a camera is freed, the pose can be adjusted with the W-A-S-D-PgDwn-PgUp and arrow keys.

Finally, in an interactive simulation, the camera can be used to collect driving commands (via the W-A-S-D keys) for use in the vehicle model.

```
driving_command = camera.GetDrivingCommand()
```

## 6.3 GPS / RTK

In MAVS-Python, an RTK sensor is simulated using empirical equations developed from Skoglund ([7]). While the MAVS-C++ library also includes a physics-based GPS / RTK model, the complexity of the inputs and calculation of the physics-based model is not exposed through the python interface.

To create a MAVS-Python RTK sensor:

```
rtk = mavs.MavsRtk()
```

There are three error parameters for the RTK sensor that must be set

```
# Set the average initial error in meters
# The RMS error will be about 20% of this value
rtk.SetError(1.0)

# Set the warmup time in seconds
# The error reduces exponentially with warmup
rtk.SetWarmupTime(60.0)

# Set the droput rate in number/hour
# Depends on the blockage of the sky
rtk.SetDroputRate(10)
```

After updating the RTK, the position and orientation (in world ENU coordinates) can be extracted by

```
position = rtk.GetPosition()
orientation = rtk.GetOrientation()
```

where "position" is an x-y-z vector in world ENU coordinates and "orientation" is a normalized quaternion.

## 6.4 Radar

A MAVS radar returns a list of targets after each update. Two parameters define the radar: the maximum range and the horizontal field of view.

```
radar = mavs.MavsRadar()
# Set max range in meters
radar.SetMaxRange(250.0)
# Set HFOV in degrees
radar.SetFieldOfView(22.5)
```

After the radar is updated, a list of targets can be accessed with

```
targets = radar.GetTargets()
```

"targets" is a list of x-y target positions in the radar sensor frame.

# Chapter 7

# MAVS-Python Vehicle

There are two options for vehicle simulation in MAVS - the default MAVS vehicle simulation and interface with the Chrono multibody-dynamics software. Chrono is not enabled by default in the MAVS-Python interface. Usage of Chrono requires that the code be built from source with proper compile flags. In this document, only the default MAVS-Python vehicle will be discussed.

The MAVS-Python vehicle uses the ReactPhysics3D library to simulate multibody dynamics. To create a new MAVS-Python vehicle

```
vehicle = mavs.MavsRp3d()
```

A vehicle file must be loaded to initialize the vehicle for simulation

```
vehicle_file = ('vehicles/rp3d_vehicles/
                forester_2017_rp3d.json')
vehicle.Load(mavs_python_paths.mavs_data_path + '/' +
             vehicle_file)
```

Example vehicle files can be found in the MAVS data folder under "vehicles/rp3d_vehicles".

After the vehicle is loaded but before the simultion begins, the initial vehicle pose can be set.

```
# Set init position in world coords
vehicle.SetInitialPosition([50.0, -50.0, 2.0])
# Set init heading in radians
vehicle.SetInitialHeading(0.2)
```

Once a vehicle is created, it can either be driven manually through a camera interface (6.2) or set to automatically follow a set of waypoints using a vehicle controller. A set of waypoints must be loaded from a file.

```
waypoints = mavs.MavsWaypoints()
waypoints_file = 'waypoints/euro_ncap_path.vprp'
waypoints.Load(mavs_python_paths.mavs_data_path + '/' +
               waypoints_file)
```

Example waypoints files are located in the MAVS data folder under the "waypoints" directory. Once the waypoints are loaded a controller must be created to automatically follow the waypoints.

```
controller = mavs.MavsVehicleController()
controller.SetDesiredPath(waypoints.GetWaypoints2D())
```

Controller properties can be set to ensure the vehicle follows the path closely and smoothly.

```
# set speed in m/s
controller.SetDesiredSpeed(5.0)
# higher values result in smoother steering
# but less able to follow tight turns
controller.SetSteeringScale(0.7)
# vehicle wheelbase in meers
controller.SetWheelbase(2.6)
# max steering angle in radians
controller.SetMaxSteerAngle(0.615)
```

Once the vehicle, waypoints, and controller have been set up, the vehicle is updated at each time step using throttle, brake, and steering commands from the controller.

```
current_pos = self.vehicle.GetPosition()
self.controller.SetCurrentState(current_pos[0],
                                current_pos[1],
                                self.vehicle.GetSpeed(),
self.vehicle.GetHeading())
current_driving_command = controller.GetDrivingCommand(dt)
vehicle.Update(env, current_driving_command.throttle,
                    current_driving_command.steering,
current_driving_command.braking,
dt)
```

# Chapter 8

# MAVS-Python Example Simulation

In this chapter, a minimal working example of a MAVS-Python is presented. This example can be copied and pasted, changing only the system path in line 6. The example creates a scene, environment, vehicle, and lidar sensor. It shows how labeled data can be saved from the lidar sensor and the vehicle can be driven through a camera window.

The first part of the script imports MAVS into Python and defines the input files for the simulation.

```
import sys
import time

# Set the path to the mavs python api, mavs_interface.py
# You will have to change this on your system.
sys.path.append(r'C:\Users\user\Desktop\mavs\src\mavs_python')

# Load the mavs python modules
import mavs_interface as mavs
import mavs_python_paths

# Set the mavs data path and the input files to load
mavs_data_path = mavs_python_paths.mavs_data_path
mavs_scene_file = "/scenes/cube_scene.json"
mavs_vehicle_file = "/vehicles/rp3d_vehicles/forester_2017_rp3d.json"
```

In the next section, a scene is created and loaded. An environment is created and the scene is added to it. Some environment parameters are also set.

```
# Create and load a scene
scene = mavs.MavsEmbreeScene()
scene.Load(mavs_data_path+mavs_scene_file)
# turn on scene labeling for annotations
scene.TurnOnLabeling()

# Create the environment
env = mavs.MavsEnvironment()
# Add the scene to the environment
env.SetScene(scene.scene)
# Set some environment properties
env.hour = 12
env.SetFog(0.03)
env.SetTurbidity(5.0)
```

After the scene is created, some sensors are added to the simulation. The first sensor is a camera that will be used to drive the vehicle. The offset of the camera sensor is set behind the vehicle so we can drive with a 3rd-person view of the vehicle. The lidar sensor is added on top of the vehicle.

```
# Create a window for driving the vehicle with the W-A-S-D keys
# Window must be highlighted to input driving commands
drive_cam = mavs.MavsCamera()
drive_cam.Initialize(256, 256, 0.0035,0.0035,0.0035)
drive_cam.SetOffset([-10.0,0.0,3.0],[1.0,0.0,0.0,0.0])
drive_cam.SetGammaAndGain(0.6,1.0)
drive_cam.RenderShadows(False)

# Create a lidar sensor for saving data
lidar = mavs_.MavsLidar('HDL-32E')
lidar.SetOffset([1.0,0,2.0],[1.0,0.0,0.0,0.0])
```

Next a vehicle is created and added to the scene at the origin.

```
# Create a vehicle and put it at the origin
vehicle = mavs.MavsRp3d()
vehicle.Load(mavs_data_path+mavs_vehicle_file)
vehicle.SetInitialPosition(0.0, 0.0, 1.0)
vehicle.SetInitialHeading(0.0)
```

Before the simulation loop starts to execute, the time step and a loop counter are initialized. The simulation and drive camera will run at 30 Hz, while the lidar sensor will run at 10 Hz. This will be achieved by only updating the lidar simulation every third time step.

```
dt = 1.0/30.0
num_lidar_spins = 0
```

Since the simulation is interactive, a timer is added on the outside of the loop to ensure that it does not run faster than real time, which would make the vehicle difficult to drive.

```
while True:
    # Timing info to be used later
    t0 = time.time()

    # Get the driving command from driving_cam window
    dc = drive_cam.GetDrivingCommand()

    # Update the vehicle simulation
    vehicle.Update(env,dc.throttle, dc.steering, dc.braking, dt)

    # Update vehicle mesh position
    env.SetActorPosition(0, vehicle.GetPosition(), vehicle.GetOrientation())

    # Advance environment time
    env.AdvanceTime(dt)

    # Update the drive cam
    drive_cam.SetEnvironmentProperties(env.obj)
    drive_cam.SetPose(vehicle.GetPosition(),vehicle.GetOrientation())
    drive_cam.Update(env,dt)
    drive_cam.Display()

    # Update the lidar sensor every 3rd step
    # effectively running at 10 Hz
    if num_lidar_spins%3==0:
        lidar.SetPose(vehicle.GetPosition(),vehicle.GetOrientation())
        lidar.Update(env,dt)
        lidar.Display()
        # uncomment the following to lines to save annotated data
        # writing a lot of file to disk will slow the sim down
        #lidar.AnnotateFrame(env)
        #lidar.AnalyzeCloud('annotated_lidar', num_lidar_spins, False)

    num_lidar_spins = num_lidar_spins + 1

    # If the simulation is running faster than real time, slow it down
    t1 = time.time()
    sleep_time = dt - (time.time()-t0)
    if sleep_time>0.0:
        time.sleep(sleep_time)
```

This simulation can be run with the command

```
>python simulation\_example_doc.py
```

The simulation can be killed from the command line with Ctrl+C. To save labeled data, the lines

```
#lidar.AnnotateFrame(env)
#lidar.AnalyzeCloud('annotated_lidar', num_lidar_spins, False)
```

can be uncommented, although it should be noted that this will slow the simulation significantly.

# Chapter 9

# Creating Photorealistic Images with MAVS

MAVS can be used to create photorealistic images like the one shown in Figure 9.1. Photorealism is achieved with path-tracing [5]. Path-tracing uses hundreds or thousands of rays per pixel, resulting in high-quality images, but also making the rendering time much slower than single-pass ray-tracing. Therefore, the path-tracer should only be used in applications where image quality is a higher priority than simulation speed.

A path-tracing camera can be created in a manner similar to other cameras in MAVS using the "model" function.

```
# 1080P HD-Camera
hd_cam = mavs.MavsCamera()
hd_cam.Model('HDPathTraced')

# Half-resolution HD-Camera
half_hd_cam = mavs.MavsCamera()
half_hd_cam.Model('HalfHDPathTraced')

# 224x224 machine vision camera
machinevision_cam = mavs.MavsCamera()
machinevision_cam.Model('MachineVisionPathTraced')
```

Alternatively, a path-traced camera can be created in which the number of rays-per-pixel and maximum pixel depth are specified.

```
res = 'medium' #options are 'low', 'medium', or 'high'
rpp = 250 # rays per pixel
```

Figure 9.1: Example of a photorealistic image created with MAVS.

```
pix_depth = 15 # max ray depth
piccut = 0.55 # must be 0-1
hd_cam = mavs.MavsPathTraceCamera(res,rpp,pixdepth,pixcut)
```

Once the camera is created, it can be used to render a scene in the same way as an ordinary camera (Section 6.2).

```
hd_cam.SetPose([0.0, 0.0, 1.5],p[1.0, 0.0, 0.0, 0.0])
hd_cam.Update(env,0.03)
hd_cam.SaveCameraImage('path_traced_image.bmp')
```

Note that many of the features enabled in the default MAVS camera such as snow, rain, and dust are not enabled with the path-tracing camera.

# Bibliography


[1] Chris Goodin, Suvash Sharma, Matthew Doude, Daniel Carruth, Lalitha Dabbiru, and Christopher Hudson. Training of neural networks with automated labeling of simulated sensor data. Technical report, SAE Technical Paper, 2019.

[2] Christopher Goodin, Daniel Carruth, Matthew Doude, and Christopher Hudson. Predicting the influence of rain on lidar in adas. *Electronics*, 8(1):89, 2019.

[3] Christopher Goodin, Matthew Doude, Christopher Hudson, and Daniel Carruth. Enabling off-road autonomous navigation-simulation of lidar in dense vegetation. *Electronics*, 7(9):154, 2018.

[4] Christopher R Hudson, Chris Goodin, Matthew Doude, and Daniel W Carruth. Analysis of dual lidar placement for off-road autonomy using mavs. In *2018 World Symposium on Digital Intelligence for Systems and Machines (DISA)*, pages 137–142. IEEE, 2018.

[5] Henrik Wann Jensen. Importance driven path tracing using the photon map. In *Rendering Techniques 95*, pages 326–335. Springer, 1995.

[6] Will Meadows, Christopher Hudson, Chris Goodin, Lalitha Dabbiru, Brandon Powell, Matt Doude, Daniel Carruth, Muhammad Islam, John E Ball, and Bo Tang. Multi-lidar placement, calibration, co-registration, and processing on a subaru forester for off-road autonomous vehicles operations. In *Autonomous Systems: Sensors, Processing, and Security for Vehicles and Infrastructure 2019*, volume 11009, page 110090J. International Society for Optics and Photonics, 2019.

[7] Martin Skoglund, Thomas Petig, Benjamin Vedder, Hans Eriksson, and Elad Michael Schiller. Static and dynamic performance evaluation of low-cost rtk gps receivers. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pages 16–19. IEEE, 2016.