Motive Bundle Documentation

*Last updated: 2021-08-09 by Shaya*

[Introduction](#_8qkq3xjrhcoh)

[System Diagram](#_kje2gbcaurao)

[Motive](#_gdo9va5acoz9)

[Initialization Steps](#_tkoup4exjhf)

[Daily Startup Steps](#_cauxh28yhstn)

[Optional: Corner Markers](#_mmgp85k035m1)

[Unique Rigid Body Markers](#_4tq6yyllf7iq)

[Naming Conventions](#_orj1b7n3im34)

[Robots](#_kh3i7jqphxs4)

[Obstacles](#_wxgpnvp6wzuc)

[Corner markers](#_om8tdojjpuek)

[The Visualizer (Python code)](#_9rqprqbt4vbn)

[Scope](#_v6oqlu9lg4m5)

[Relevant Files](#_5pzmyh3hrx9j)

[Map File](#_ejfov28vd21j)

[Scenario File](#_ynw3y3bid7r9)

[End Location File](#_demkriir9l07)

[Paths File](#_6i3j9cynj84z)

[Plan File](#_frogkz49cd32)

[Relevant Variables](#_sm46txsmhcrm)

[Buttons on the Visualization](#_j2e8z2d9u09f)

[Make .SCEN](#_1koch7ymcevd)

[Load from .SCEN](#_6l7dw6elxnk5)

[Run Planner](#_5g96bg83ti3)

[Choose End Locs](#_cdrl2y6rfzi9)

[UDP Broadcasting](#_2ayqz7wnz38m)

[How to Run a Simple Demo](#_xnyheuuooz08)

[What is a demo?](#_t7xzor2r6igb)

[Steps for a Simple Demo](#_qzjibpiytszt)

[Issue Documentation](#_dshvq0d1kf59)

[Quaternion conversion issue](#_ae128bfrf3eq)

[UDP Broadcasting Data Jam](#_dlse90kostrs)

# Introduction

This document outlines important information about CRL’s [Motive Visualization](https://github.com/nirgreshler/natnet-motionclient.git) project. The objective of the system is to offer a visual representation of the robots and obstacles in the arena, to process and send live position and rotation data to the Ubuntu computer in order to create closed-loop demos, and, most importantly, to allow for the application of Multi-Agent Path-Finding (MAPF) algorithms to live scenarios.

In this document, you’ll find **basic** instructions on using Motive (more detailed documentation [here](https://docs.google.com/document/d/14xVPKJvb1QKJlfr7BtcffD3xK34WtcWJHHqQjFuFkb0/edit?usp=sharing) and [here](https://v22.wiki.optitrack.com/index.php?title=Motive_Documentation)), instructions on using our Visualizer to interact with Motive and run a demo, and documentation of problems we have encountered and how they were handled.

# [System Diagram](https://drive.google.com/file/d/1W80WzPDVS30EN8Raq1rpQlVar0nzf0DM/view?usp=sharing)

## 

# Motive

## Initialization Steps

The following steps should be followed once so that you don’t have to do it again every day.

1. **Prepare robots and obstacles.** Stick the retroreflective markers in Motive’s kit to the rigid bodies using the adhesive circles provided. Refer to [guidelines below](#_4tq6yyllf7iq) for marker placement conventions and the [Motive Mini Documentation](https://docs.google.com/document/d/14xVPKJvb1QKJlfr7BtcffD3xK34WtcWJHHqQjFuFkb0/edit#heading=h.lkoc73baoeoh) for more detailed information.
2. **Introduce Rigid Bodies in Motive.** Referring to [Motive Mini Documentation,](https://docs.google.com/document/d/14xVPKJvb1QKJlfr7BtcffD3xK34WtcWJHHqQjFuFkb0/edit#heading=h.9x3wh0ufbtbz) use the Motive application to create a rigid body corresponding with each obstacle and robot. Name each rigid body according to the [convention](#_orj1b7n3im34) outlined below.

## Daily Startup Steps

The following steps should be followed each time you wish to use the system.

1. **Place robots and obstacles in the arena and verify that the system recognizes them.** Once the rigid bodies are placed, as long as the relative placement of the retroreflective markers has not changed, Motive should automatically recognize and label each body. If there are any issues or inconsistencies, re-initialize the rigid body.
2. **Turn on all of the robots.** Use the small switch they have, and make sure they play their startup sound a few seconds after you start them.
3. **Initialize angles.** Turn each robot so that its wheels face **right** (away from the door; towards the window). On Motive, open the Builder pane , and click on the “Edit” pane. Under the “Orientation” section, click “Reset.”

## Optional: Corner Markers

As an alternative to using [parameters](#_sm46txsmhcrm) to control the edges of the robots’ arena, you can use physical corner markers placed in the arena.

**Directions**: Place the four small cardboard boxes in the desired four corners of the arena, taking care to match the label on the box to the corner you are placing it in (e.g. the box labeled “BOTTOM RIGHT” should be placed nearest to the computer you are working on). Make sure all four rigid bodies are labeled as the [convention](#_om8tdojjpuek) defines.

The visualizer will pick up on the locations of the four boxes and restrict the arena to match.

## Unique Rigid Body Markers

All rigid bodies should have unique marker configuration to avoid confusion in the system (especially if there are many similar robots in the system). Note that each rigid body must have at least 3 markers to be recognized by the system and should not have more than 8. Vary size and height of markers for uniqueness.

**Robots**: Make marker patterns as unique as possible, such that the center of the markers somewhat aligns with the center of the robot.

**Obstacles**: Place a marker on each corner/vertex of the obstacle and then create a unique pattern on the inside, or add at least one identifying marker.

## Naming Conventions

The naming conventions defined below were developed alongside the Visualizer code such that the visualizer is able to recognize and interpret the name of each robot, corner marker, and obstacle as is intended.

### Robots

**Name**: [name]-[id] (ex. Ruby-1, Rosie-2, …)

**Streaming ID**: Robots’ unique Streaming IDs should be in the 100’s (ex. 101, 102, 103, etc..). A robot’s ID should be 100 + the number at the end of its name (ex. Ruby-1 and 101).

### Obstacles

**Name**: Obstacle-1, Obstacle-2, …(must have “obst” in it)

**Streaming ID**: Obstacle Streaming IDs should be in the 200s (ex. 201, 202, …). Their ID should be 200 + the number at the end of their name (ex. Obstacle-1, 201).

### Corner markers

There are only four corner markers; they should be named and numbered as defined below. Directions are relative to the perspective of somebody sitting in the chair at CRL that overlooks the robot area, where “top” is farther away from the viewer and “bottom” is closer.

|  |  |  |
| --- | --- | --- |
| Where you should put it | Name | Streaming ID |
| Top Left | Corner-TL | 301 |
| Top Right | Corner-TR | 302 |
| Bottom Left | Corner-BL | 303 |
| Bottom Right | Corner-BR | 304 |

# The Visualizer (Python code)

## Scope

The purpose of the Visualizer is to facilitate Multi-Agent Path Finding demos in CRL. It starts with receiving data broadcast by the Motive application, which lives on the same computer as the Visualizer itself. On the other end, it sends live position/rotation data to the Ubuntu computer via UDP broadcast, and applies MAPF algorithms to generate paths for each robot to its desired location, which it also sends to the Ubuntu computer via PSCP (a Putty application).

## Relevant Files

### Map File

A .map file is a file generated by the visualizer that represents the robots’ environment. It is a grid composed of ‘@’ and ‘.’ characters, where the former represents cells occupied by obstacles and the latter represents empty spaces (or robots). This file is part of the path planner’s input.

**Default Name:** map.map

### Scenario File

Similarly, a .scen file is also generated by the visualizer for the purpose of serving as input for the path planner. Each entry (line) represents one robot in the environment.

For more detailed information on the content and format of .scen files refer to [this documentation](https://movingai.com/benchmarks/formats.html). For benchmarks of both .MAP and .SCEN files, refer to [this page](https://movingai.com/benchmarks/index.html).

**Default Name:** scenario.scen

**Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bucket | Map width | Map height | Start x | Start y | End goal x | End goal y | Optimal length |

### End Location File

A tab-separated .TXT file created by the user that allows the user to choose the robots’ end locations using the [“File” button](#_cdrl2y6rfzi9).

**Default Name:** end\_locations.txt

**Format:**

|  |  |  |
| --- | --- | --- |
| Robot ID | X | Y |

### Paths File

A .TXT file generated by the MAPF algorithm that defines the paths calculated for each robot.

**Default Name:** paths.txt

**Format:**

Agent [ID]: ([x],[y])->([x],[y])->...->

### Plan File

A .TXT file that carries the same information as the aforementioned Paths file, except presented in a format that is readable for Hadar’s ROS code.

**Default Name:** algorithm\_output.txt

**Format**:

schedule:

agent0:

* + x: [x]

y: [y]

t: [time step]

* + x: [x]

y: [y]

t: [time step]

* + x: [x]

y: [y]

t: [time step]

agent1:

* + x: [x]

y: [y]

t: [time step]

* + x: [x]

y: [y]

t: [time step]

## Relevant Variables

x\_dim, y\_dim (found in \_\_init\_\_() in Grid.py): specifies the width and height of the *active* robot arena in meters; the greater arena will always be 12x12m; default 10x5m.

cell\_size (found in \_\_init\_\_() function in Grid.py): size in square meters of each grid cell. Defaults to 1; make it smaller for higher “resolution”

tolerance (found in add\_body() function in Grid.py): how “strictly” a robot needs to adhere to its cell in order for the system to recognize it as a robot in a cell. Options are:

0 - zero tolerance, entire robot must be on one cell

1 - one marker may be outside of the cell

2 - majority markers can be on one cell

defaults to 1.

dr (found in getBlockedCells() in Grid.py): the intervals at which samples will be taken in between markers when determining the cells that are covered by an obstacle. Setting this variable to a lower value will make the system more thorough (less likely to miss a small corner or vertex).

## Buttons on the Visualization

### Make .SCEN

Generates [Map](#_ejfov28vd21j) and [Scenario](#_ynw3y3bid7r9) files; displays end locations according to robot ID on the grid visualization.

### Load from .SCEN

Loads end locations for robots from an existing [Scenario](#_ynw3y3bid7r9) file. Should only be used if the arrangement of rigid bodies in the arena has not changed since the desired Scenario file was generated.

### Run Planner

Using existing [Scenario](#_ynw3y3bid7r9) and [Map](#_ejfov28vd21j) files, runs the MAPF algorithm and generates a [Paths](#_6i3j9cynj84z) file; converts the Paths file to a [Plan](#_frogkz49cd32) file. Displays paths on screen. Only run this when you are sure that the Scenario and Map files are consistent with the real-life arena.

### Choose End Locs

Loads end locations for robots from an existing tab-separated [End Location file](#_demkriir9l07) and creates a [Scenario](#_ynw3y3bid7r9) file from them. (Different from “Load” feature in that the File feature allows the user to easily choose end locations, doesn’t require existence of a previously generated Scenario file, and doesn’t involve start locations).

## UDP Broadcasting

The Visualizer implements a UDP server to send data between the computer with Motive and the computer with Ubuntu. It sends position and rotation data from the robots only in the form of Python dictionaries.

Steps:

1. Connect both computers to the CIS network.
2. In udp\_server.py, make sure the variable “self.localIP” is set to the computer’s IP address (should be what you get when you type in google “what is my IP address”)
3. In udp\_client.py, make sure the first parameter passed into serverAddressPort on line 9 matches the IP address in Step 1.
4. Begin by broadcasting from the SERVER computer first by running main\_UDP.
5. Then, receive data from upd\_client.py or from your desired ROS node.

Tips and common issues we have encountered:

* If the data being sent each time is an empty list that looks like this: “[ ]”, that means UDP’s connection works fine, but the server is simply not sending data. Try to ensure that Motive is broadcasting data.
* Every time the *client* stops requesting data, the server will also stop sending it until both machines start fresh.
* If you are stuck, WireShark sometimes helps.

# How to Run a Simple Demo

## What is a demo?

A demo includes placing robots, obstacles, and corner markers in various locations (as desired) around the arena, using the Visualizer to apply a MAPF algorithm and generate a plan, and then applying that plan to finally watch the robots move.

## Steps for a Simple Demo

1. **Initialize the arena and Motive.** Refer to the [Initialization Steps](#_tkoup4exjhf), which should be followed once, and the [Daily Startup Steps](#_cauxh28yhstn), which should be followed each time you run a demo.
2. **Start the Visualizer** by running main\_planner.py.
3. **Create Scenario.** To generate a Map and Scenario, click “Make .SCEN.” Continue clicking it until you are happy with the end locations that show up on the grid.
4. **Run Planner.** Click “Run Planner.” The plan will be automatically sent to the Ubuntu computer.
5. **Begin UDP broadcast.** Exit main\_planner.py and run main\_UDP.py.
6. [Hadar fill in the steps for ROS]
7. Watch the robots move!

# Issue Documentation

#### Quaternion conversion issue

Date: 4 August 2021

Description: Motive’s API only broadcasts rotation data in quaternion format, while our system needs the Yaw (rotation about the y-axis) value. We converted using [this](https://automaticaddison.com/how-to-convert-a-quaternion-into-euler-angles-in-python/) function, but the conversion yielded Euler data that did not match the Euler values reported on Motive’s user interface. We found that we needed to swap the interpretation of the parameters in the conversion function such that:

self.x = quat.w

self.y = quat.x

self.z = quat.y

self.w = quat.z

Where quat was the quaternion object passed in from Motive and self was the function’s own values.

#### UDP Broadcasting Data Jam

Date: 18 August 2021 (commit e0c5be9aacf7a645f5979ae5d6a2d03abdbf4183)

Description: UDP broadcast used a LIFO queue by default (or maybe Nir set it up this way); it would push to the queue every data update it received from Motive and then pull off the item that was sitting there the longest. Because data was pulled at a slower rate than it was pushed, it caused a backup in data and an increasing delay/misalignment between reality and what was being broadcasted. To solve the issue, we edited the file udp\_server.py so that the queue now has a maximum of one item at all times; if the program is ready to push new data but there is already something there, it will replace the old item with the new one.

#### Kill UDP/TCP process

Date: 10 November 2021

Description: Sometimes, terminating the program that runs UDP server (say, main\_udp.py) doesn’t end gracefully, resulting in successive run of the program not able to start the UDP server again.   
The fix can be to kill the remaining UDP(/TCP) process that blocks our port:

Open the terminal, and run:  
netstat -ano | findstr :*port\_number*

We’ll get the PID of the process (e.g. 11704) and then we can kill it, by.:

taskkill /pid 11704 /F

Ref: <https://stackoverflow.com/questions/8688949/how-to-close-tcp-and-udp-ports-via-windows-command-line>