DRONE SIMULATION

UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLIARD

www.utbm.fr

Université de Technologie de Belfort-Montbéliard
Laboratoire CIAD
13 rue Ernest Thierry Mieg
90010 Belfort cedex, France

Document Reference	2021_IA51_7
Date of publication	2021-06-10
Members of the Group	LICATA Enzo enzo.licata@utbm.fr LIEFFROY Théo theo.lieffroy@utbm.fr FARDEAU-SPIRE Malek malek.fardeau@utbm.fr

CONTENT

Content 3
1. Simulator Architecture 3
2. Environment Implementation 3
2.1. Environment Objects 3
2.2. Environment Structure 3
2.3. Environment Dynamics 3
2.4. Agent Perception 4
2.5. Agent Actions or Influences 6
4. Agent Behaviors 6
5. User Interface 10
6. Simulation Results 11
6. Challenges encountered 11

1. SIMULATOR ARCHITECTURE

Our simulator architecture is based on the environment provided by the AirSim Simulator from Microsoft running on the Unreal Engine. We then communicate with it using the provided Java to Airsim API to retrieve the perceptions of the drones and send the movement commands. The programming language used for the agents in this project is SARL, a general-purpose agent-oriented language working on the Eclipse IDE. So for this project to run, you can use the provided executable that will execute the Sarl Program and also setup the AirSim settings for you, but you need to have AirSim installed on your computer with an environnment setup (you can use the default one Blocks.sln)

2. ENVIRONMENT IMPLEMENTATION

2.1. ENVIRONMENT OBJECTS

The simulator's environment is composed of several objects such as buildings (cubes), trees (cones) and monuments (spheres) which simulate a real-life environment. We are using the basic scene of the AirSim simulator to which we have added and positioned our own obstacles.

2.2. ENVIRONMENT STRUCTURE

The environment generated by AirSim is represented with a 3D reference frame in (X, Y, Z) coordinates used by Unreal. We then recover data such as the drone's coordinates (X, Y, Z) or orientation (Quaternion that we convert to roll pitch and yaw) thanks to the APIs provided by AirSim.

2.3. ENVIRONMENT DYNAMICS

The Airsim Environment is already provided with the influence of gravity taken into account, but we can also optionnaly add other dynamics such as wind simulation or other weather effects

2.4. AGENT PERCEPTION

All the agents perceptions are retrieved in the loop of the SimulationControllerAgent :

```
/** Retrieve the perceptions for a single agent */
private def retrieveAgentPerceptions(address : Address, name: String) : void {
    val state : MultirotorState = name.multirotorState
    val pose : Pose = name.simGetObjectPose
    val kinematics : KinematicsState = name.simGetGroundTruthKinematics
    val lidar : LidarData = name.getLidarData

    influenceReactionStrategy.perceptionReceived(address) [
        emit(new MultirotorStatePerception(state))[it == address]
        emit(new LidarDataPerception(lidar))[it == address]
        emit(new SimPosePerception(pose))[it == address]
        emit(new SimGroundTruthKinematicsPerception(kinematics))[it == address]
    ]
}
```

Indeed, this agent will loop calls to the AirSim API to retrieve the perceptions data such as Pose (position + orientation) with simGetObjectPose or LidarData (Array of float of the coordinates of the points detected) with getLidarData :

```
skill AirSimSimulationPerception implements SimulationPerceptionCapacity {

    val loop : EventLoop = EventLoop.defaultEventLoop()
    val rpcClient : Client
    val rpcLibClient : RpcLibClientBase

    new() {
        this("127.0.0.1", 41451)
    }

    new(ipAddress: String, port: Integer) {
        rpcClient = new Client(ipAddress, port, loop)
        rpcLibClient = rpcClient.proxy(typeof(RpcLibClientBase))
    }

    def simGetObjectPose(objectName : String) : Pose {
        synchronized (this.rpcLibClient) {
            return rpcLibClient.simGetObjectPose(objectName)
        }
    }


    def getLidarData(vehicleName : String) : LidarData {
        synchronized (this.rpcLibClient) {
            return rpcLibClient.getLidarData("Lidar", vehicleName)
        }
    }
```

Once the perceptions are retrieved from AirSim, the SimulationControllerAgent then sends the corresponding event for the agents concerned to retrieve their own data.

For example, when the SimPosePerception event is received by our drone agent, it updates its own position and orientation value (after converting it from Quaternion to Euler Angles):

```
//Update SARL position and orientation when a PosePerception is received from Airsim
on SimPosePerception {
    val positionPerceived : Pose = occurrence.pose //We retrieve the position perceived from Airsim
    this.position = new Vector3(positionPerceived.position)
    val quaternion : Vector4 = new Vector4(positionPerceived.orientation.x, positionPerceived.orientation.y,positio
    this.orientation = quaternion.toEulerAngles()
}
```

The same happens when LidarDataPerception event is received, the drone updates its own lidarData value with the lidarPointCloud propagated by the event:

```
// Update SARL LidarData when a LidarDataPerception is received from Airsim
on LidarDataPerception {
    val lidarPointCloud : float[] = occurrence.lidarData.pointCloud
    this.lidarData = lidarPointCloud
}
```

If we use the ReachLocationWithDronesAndObstaclesAvoidance behavior, each drone also sends its position and velocity to the other ones when they have received them using the NotifyKinematics and NotifyPose events:

```
on SimPosePerception {
    val positionPerceived : Pose = occurrence.pose //We retrieve the position perceived from Airsim
    this.position = new Vector3(positionPerceived.position)
    val quaternion : Vector4 = new Vector4(positionPerceived.orientation.x, positionPerceived.orientation.y,positionPerceived
    this.orientation = quaternion.toEulerAngles()

    emit(new NotifyPose(positionPerceived))
}
```

```
// Receive the others agent position
on NotifyPose [!isMe(occurrence.source)] {
    if (neighbours.containsKey(occurrence.source)) {
        neighbours.put(occurrence.source,
            new ExternalDroneState(neighbours.get(occurrence.source), occurrence.pose))
    } else {
        neighbours.put(occurrence.source, new ExternalDroneState(occurrence.pose, null))
    }
}

//Receive the others agent velocity
on NotifyKinematics [!isMe(occurrence.source)] {
    if (neighbours.containsKey(occurrence.source)) {
        neighbours.put(occurrence.source,
            new ExternalDroneState(neighbours.get(occurrence.source), occurrence.state))
    } else {
        val nullPose : Pose = null
        neighbours.put(occurrence.source, new ExternalDroneState(nullPose, occurrence.state))
    }
}
```

2.5. AGENT ACTIONS OR INFLUENCES

For the agent actions in return, it works the opposite way. The agent generates a new action event, either MoveByVelocity or RotateToYaw, which are then received in the SimulationControllerAgent :

```
on MoveByVelocity {
    debug("MoveByVelocity (begin)")
    val body : String = affectationStrategy.affectOrGetBody(occurrence.source, [a, n|onAffectation(a, n)])
    influenceReactionStrategy.influenceReceived(occurrence.source) [
        body.moveByVelocity(occurrence.vx, occurrence.vy, occurrence.vz, occurrence.duration)
    ]
    debug("MoveByVelocity (end)")
}

on RotateToYaw {
    debug("RotateToYaw (begin)")
    val body : String = affectationStrategy.affectOrGetBody(occurrence.source, [a, n|onAffectation(a, n)])
    influenceReactionStrategy.influenceReceived(occurrence.source) [
        body.rotateToYaw(occurrence.rot)
    ]
    debug("RotateToYaw (end)")
}
```

The SimulationControllerAgent then takes care of sending those orders via API calls to Airsim:

```
override moveByVelocity(vehicleName : String, vx : Float, vy : Float, vz : Float, duration : Float) : void {
    synchronized (this.rpcLibClient) {
        rpcLibClient.moveByVelocityAsync(vx, vy, vz, duration, DrivetrainType.MAX_DEGREE_OF_FREEDOM, new YawMode(),
            vehicleName)
    }
}
```

4. AGENT BEHAVIORS

Our drone agents implement the ReachLocationWithObstacleAvoidance behaviour that we built aimed at making them able to go from a starting location to a target location while avoiding the obstacles on their way. This behaviour is composed of multiple variables to store all the data we retrieve from the perceptions received from Airsim and sent to us through events by the SimulationControllerAgent. All this data is then used to make decisions based on a decision tree that then leads to an action event that we will emit in return for the SimulationControllerAgent to receive and transmit back to Airsim to make it happen in the Unreal Environment. We then loop through all this, receiving perceptions and sending actions back to Airsim in real time to enable our drones to always navigate their environment. We also built a ReachLocationWithDronesAndObstacleAvoidance behaviour that uses the previously built computeSlidingForce() function on top of the LidarData to know and avoid collisions with other drones.

When the simulation is initialized, the SimulationControllerEvent sends the SimulationInitialized event to all the agents. When this event is received by the drones, we can start sending actions to AirSim. The first thing we do is to send a Takeoff event to the

SimulationControllerAgent so that it is propagated back to Airsim to make the drones takeoff. We then copy our initial location to be able to go back and we generate a new target location using the function generateTargetLocation(). When the initialization is done, we can then launch the loop of decision making that generates a new MoveByVelocityEvent every iteration using the results of the computeForce() function:

```
//For loop behavior when the simulation starts
on SimulationInitialized {
    emit(new Takeoff) //Emit takeoff event to notify the SimulationControllerAgent

    in(5000) [ // Initial delay before starting
        this.initiallocation = new Vector3(this.position.x,this.position.y,0f)
        this.targetlocation = generateTargetLocation() //We generate a random target location
        info("My target location is : " + this.targetlocation.x + ", " + this.targetlocation.y + ", 0")

        every(periodMs) [ //Period at which we reevaluate the drone actions
            val movementForce : Vector3 = computeForce() // Create a force vector for the movement

            emit(new MoveByVelocity(movementForce.getX(), movementForce.getY(), movementForce.getZ(), periodMs/1000f))
        ]
    ]

}
```

Once the drone has taken off, the decision tree then computes its data to choose the next action to apply. First, we test if the drone reached the X and Y coordinates of the target location within a distance margin error:

```
//Function that computes the force to move the drone
def computeForce() : Vector3 {
    //We copy the variables to avoid problems with parallel execution
    var result : Vector3 = new Vector3
    val position : Vector3 = new Vector3(this.position)
    val orientationz : float = Math.toDegrees(this.orientation.z as double)as float

    info("My position : " + position.x + ", " + position.y + ", " + position.z + ". My orientation : " + orientationz)
    if ((position.x < (this.targetlocation.x + distanceError)) && //Test if we reached the target location
    (position.x > (this.targetlocation.x - distanceError)) &&
    (position.y < (this.targetlocation.y + distanceError)) &&
    (position.y > (this.targetlocation.y - distanceError)))
    {
```

If the X and Y coordinates correspond to the target location, then we must test if the Z coordinate is also corresponding. If the Z coordinate is too high that means the drone needs to descend (landing) by generating the appropriate velocity thanks to generateVelocity() function:

```
if(position.z < (1 - distanceError)) { //If we need to descend
    info("I'm landing")
    result = generateVelocity(0f,0f,1f)
}
```

However, if the Z location correspond to the target location's one, that means the drone reached his target location, therefore, we need to generate a new location. There are two possibilities: either we are currently at home location, so we need to generate a new

random location, or we just reached a random target location, so we need to generate the new target location with the home location coordinate in order to go back to base:

```
else { //If we are arrived, we generate a new location
    if((this.targetlocation.x == this.initiallocation.x) && (this.targetlocation.y == this.initiallocation.y)) { //If we are bac
        info("I'm back to base, I go to a new location")
        this.targetlocation = generateTargetLocation()
    }
    else {
        info("I'm arrived, I go back to base")
        this.targetlocation = this.initiallocation
    }
}
```

Once the drone reached a target location it needs to ascend. Same operation as the descend function, we generate the appropriate velocity with generateVelocity():

```
else if (position.z > -2) { //If we need to ascend
    info("I ascend")
    result = generateVelocity(0f,0f,-1f)
```

If we are using the ReachLocationWithDronesAndObstaclesAvoidance behaviour, the drones are also sending each other their location and velocity. In this case, before checking the LidarData, we first use the functions already provided in the first iteration of the behaviours of the project, the computeSlidingForce() function renamed here in computeDronesToAvoid, to see if a drone will be in the path of another one based on its target location, current position, velocity, and acceleration . If this is the case, we use the resulting force to avoid the drone :

```
val droneToAvoid : Vector3 = computeDronesToAvoid()
if (droneToAvoid !== null ) {
    if ((droneToAvoid.x != 0f) || (droneToAvoid.y != 0f) || (droneToAvoid.z != 0f)) {
        result = droneToAvoid
        info("I avoid a drone with a force : " + result.x + ", " + result.y + ", " + result.z)
        return result
    }
}
```

While flying the drone might need to avoid an obstacle, to do so, whenever the lidar detect an obstacle within the close range of the drone, we compute the coordinates of the mean point to avoid (among 5000 points) with the computePointToAvoid function and we give the drone the order to move at the exact opposite of this point with a strong value :

```
else {
    val pointToAvoid : Vector3 = computePointsToAvoid()
    if ((pointToAvoid.x != 0f) || (pointToAvoid.y != 0f) || (pointToAvoid.z != 0f)) { // If we need
        var distance : Vector3 = new Vector3()
        distance = pointToAvoid - position
        var velocity : Vector3 = generateVelocity(distance.x,distance.y,distance.z)
        result = new Vector3(-2f * velocity.x, -2f * velocity.y, -0.5f * velocity.z) //We want to g
        info("I avoid the obstacle with a force : " + result.x + ", " + result.y + ", " + result.z)
    }
```

To trigger the forward procedure, we need to give the right angle of orientation to the drone. The angle is calculated thanks to the *atan2()* maths function which provides an angle of orientation between -180° and 180°:

```
else {
    var distance : Vector3 = new Vector3()
    distance = this.targetlocation - position
    var angletohave : float = Math.toDegrees(Math.atan2(-distance.y as double, -distance.x as double)) as float
```

Once the angle has been set up, we need to check if it is correct, and if so, we can trigger the forward procedure with generateVelocity() function, otherwise we keep rotating the drone:

```
if (((orientationz < (angletohave + degreeError)) && (orientationz > (angletohave - degreeError)))) { //
    info("I move in direction of the target location")
    result = generateVelocity(distance.x, distance.y, 0f,true)
}
else { // Else we rotate to be in line with the target
    info("I rotate of to be in line with the target location : " + angletohave)
    emit(new RotateToYaw(angletohave))
}
```

We now have all the decisions that our drone can take in order to navigate its environment correctly.

How does the computePointToAvoid function work?

Once the lidar sense an obstacle, we compute the coordinates of the mean point among all the detected points, and we reset the Lidar's data.

```
def computePointsToAvoid() : Vector3 {
    var result : Vector3 = new Vector3(0f, 0f, 0f)

    val lidarPointCloud : float[] = this.lidarData
    var lidarSize : int = lidarPointCloud.size
    if (lidarSize >= 3) { // If we detected an obstacle with lidar
        var meanPoint : float[] = #[lidarPointCloud.get(0),lidarPointCloud.get(1),lidarPointCloud.get(2)]
        for (var i = 3;i<lidarSize;i++) { //We compute the mean of all the points detected
            meanPoint.set(i%3,meanPoint.get(i%3)+lidarPointCloud.get(i))
        }
        lidarSize = lidarSize/3
        val pointToAvoid : Vector3 = new Vector3(meanPoint.get(0)/lidarSize, meanPoint.get(1)/lidarSize, meanPoint.get(2)/lidarSize)
        info("I've go an obstacle to avoid : " + pointToAvoid.x + ", " + pointToAvoid.y + ", " + pointToAvoid.z)
        this.lidarData = null // To not recompute old LidarData
        result = pointToAvoid
    }
    return result
}
```

How does the generateVelocity function work?

Based on the distance to a target point, we generate values with a ratio of X, Y, and Z divided by the sum off all values. We then multiply this percentages by a constant value : velocityScalingValue, to ensure that the velocity we give never exceeds a too high value. For example, if we give a vector with x=0, y=0 and z=1, the velocity of z will be the max value velocityScalingValue while x and y will stay at 0. If the diminishWhenClose parameter is true, we give way lower values to be able to reach our targetLocation without going too far :

```
def generateVelocity(x : float,y : float, z : float, diminishwhenclose : boolean = false) : Vector3 { //This function is used to gene
    var result : Vector3 = null

    val sum : float = Math.abs(x) + Math.abs(y) + Math.abs(z)
    if ((sum < 10*this.distanceError) && (diminishwhenclose == true)) { //If we are really close to the target location we want to go
        result = new Vector3(x/sum,y/sum,z)
    }
    else { //Else we scale the values
        result = new Vector3((x/sum)*this.velocityScalingValue,(y/sum)*this.velocityScalingValue,(z/sum)*this.velocityScalingValue)
    }
    return result
}
```

How does the generateTargetLocation function work?

In order to simulate new delivery locations, we generate random coordinates among a known range :

```
def generateTargetLocation() : Vector3 {
    return new Vector3(((Math.random() * randomTargetLocationRange * 2) - randomTargetLocationRange) as float,((Math.random() * rand
}
```

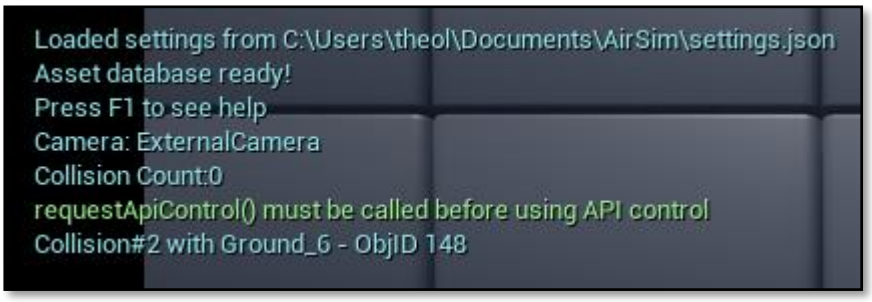
5. USER INTERFACE

Once the program starts, the user is asked to enter the number of drones to be generated, in an input box. This number will be used to generate a settings.json file in the right computer directory in order to create the right number of drones in AirSim then it will be asked to launch AirSim with another popup window before resuming the program to make sure everything is launched in the right state.

6. SIMULATION RESULTS

Measure 1: Collisions

AirSim provides a collision counter that can notify us when a drone has a collision.



6. CHALLENGES ENCOUNTERED

While developing our project we encountered several problems and challenges that we needed to solve to advance further in the development.

First, we had a lot of problems making SARL to work in cunjunction with AirSim. We had to use the AirSim-To-Java API made by Alexandre Lombard in Kotlin, that we then needed to compile using Gradle. But this API was not working straight up and we had to do a lot of troubleshooting to make it work and exchange with AirSim, including reverting our SARL Version to 10.1, our Java to JDK 11, and changing the way the simulation controller agent works to realTimeSimulationStrategy since the fixedStepSimulationStrategy that it was using before that paused the simulation after every step to have the time to properly receiver perceptions, compute them and send actions back to AirSim was not working anymore with the latest AirSim version.

Then a member of our group, Malek, lost his computer in the middle of May and had to install everything again while working on a computer that was not really able to handle the agents on SARL and the simulation in parallel in real time. This problem and the fact we were forced to use the AirSim API in RealTime is part of the reason we decided to focus more on the decision making of a single agent, like going from point A to point B while avoiding obstacles, more than the communication between multiple agents, avoiding each other, etc ... since we were not really able to run the environment with more than one agent since the program would lag and API calls would get lost making it impossible to debug.

There were also a bunch of times were the AirSim-To-Java API was not working with the current AirSim version or the functionnalities were not entirely implemented in the first place and we had to edit and compile the Kotlin Code to make our drones work. For example, the Rotate functions were not working, and we had to change the API calls sent to AirSim, and write the code to be able to send them with the SimulationControllerAgent. The LidarDataPerception was also not working with the latest AirSim version since a new item was added and we had to manually add it to the rpc Client calls to AirSim, before compiling it with gradle which was really hard to troubleshoot, while also adding the Capacity to the

SimulationControllerAgent in our code, and tweaking the settings.json of AirSim to add Lidar capabilites to the drone in the way we wanted. Therefore, you can find 2 Kotlin files in our github repository that you first need to replace in the AirSim-to-Java API before compiling it with gradle in order to make our version work.

We also got issues making our decision-making code work, for example we had some troubles computing the angle of orientation for the drones to take and also comparing it with the retrieved drone orientation data to allow them to trigger the forward procedure. We in fact had to implement a way to convert the retrieved orientation data of the drone in quaternion to euler angles to be able to compute it. We also first used the simple *atan()* maths function to compute the angle to take, however, the calculated angle after the rotation wasn't correct since it was limited to a value between –90° and 90°. To solve this problem, we changed to a version using the *atan2()* maths function, which enabled us to compute our angle in 360° directions and it then worked perfectly fine.