RflySim3D常用场景控制接口实验原理

3.文件目录

例程目录：[安装目录]\RflySimAPIs\3.RflySim3DUE\0.ApiExps\e6_RflySim3DCtrlAPI

序号	实验名称	简介
1	三维场景交互接口场景控制python接口验证实验	在进行仿真时， Python函数通过调用RflySim3D的命令接口函数或蓝图接口函 实现包括发送命令、更新无人机状态、附加无人机等操作。
2	三维场景交互接口python获取飞机，物体， 相机信息接口getCamCoptObj验证实验	通过python接口获取飞机、物体和相机的信息。
3	三维场景交互接口bat脚本加载模型实验	利用bat脚本和Python脚本快速布置RflySim3D场景。
4	三维场景交互接口RflySim3D加载txt脚本接口RflyLoad3DFile实验 （通过python sendUE4Cmd调用）	熟悉创建物体和移动物体指令， 通过读取文件的形式操作RflySim3D场景。
5	三维场景交互接口移动物体创建实验（通过python sendUE4Pos创建）	通过python接口创建物体， 并通过循环发送UDP不断调整物体位置。

序号	实验名称	简介
6	三维场景交互接口python和simulink通过udp通信验证实验	Simulink发送数据到python， 验证控制RflySim3D的UDP通信接口。
7	三维场景交互接口视角调整实验（python）	熟悉通过python调整UE观察视角接口。
8	三维场景交互接口爆炸特效触发实验（python 通过sendUE4ExtAct触发）	验证蓝图模型的爆炸特效接口。
9	三维场景交互接口飞行器标签信息设置显示实验（python）	通过调用python 接口， 创建目标以及设置目标的标签属性等。
10	三维场景交互接口RflySim3D更新状态获取接口（python getUE4Pos）	通过平台提供的python接口获取RflySim3D 内所有动态创建物体位置、碰撞数据。

总体说明

UE4CtrlAPI.py（常用场景控制函数库）

UE4CtrlAPI 是一个 Python 库，它提供了一系列接口来控制 RflySim3D
模拟环境中的对象。它可以用来向模拟器发送命令和请求，例如移动相机，改变天气，或添加障碍物。该类还允许接收模拟器的数据，例如飞行器的位置和速度，对象的碰撞状态，或相机的图像数据。该模块使用两个
UDP 套接字进行通信：一个用于发送广播消息，一个用于接收模拟器的数据。

UE4CtrlAPI类的构造函数

UE4CtrlAPI.py模块中的“UE4CtrlAPI”类中包含主要的场景控制接口，可将各种消息封装为UDP然后发送出去，同时还可以接收RflySim3D发送场景中的各类UDP消息。其构造函数如下：

constructor function

def *_init_*(self, ip='127.0.0.1'):

self.ip = ip

self.udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # Create
socket

self.udp_socket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

self.udp_socketUE4 = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # Create
socket

self.udp_socketUE4.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)

self.inSilVect = []

self.inReqVect = []

self.stopFlagUE4=True

self.CoptDataVect=[]

self.ObjDataVect=[]

self.CamDataVect=[]

self.hasMsgEvent = threading.Event()

self.trueMsgEvent = threading.Event()

这个类有四个属性：

ip主机的IP地址，self.udp_socket
用于发送广播消息，它被设置为支持广播，self.udp_socketUE4
用于接收来自RflySim3D的数据，它被设置为支持地址重用，stopFlagUE4:
一个布尔标志，表示是否停止接收RflySim3D的数据。

inSilVect: 存储要发送给RflySim3D的静默命令的列表

inReqVect: 存储要发送给RflySim3D的请求命令的列表

CoptDataVect: 存储从RflySim3D接收的关于飞行器对象的数据的列表

ObjDataVect: 存储从RflySim3D接收的关于其他对象的数据的列表

CamDataVect: 存储从RflySim3D接收的关于相机对象的数据的列表

self.hasMsgEvent 和 self.trueMsgEvent
是用于线程间通信的事件对象。self.hasMsgEvent
表示是否有消息要发送给RflySim3D，self.trueMsgEvent
表示是否收到了RflySim3D的回复。这两个事件对象可以用于实现同步或异步的消息传递机制。

UE4CtrlAPI类的调用方法

利用“UE4CtrlAPI.py”库文件中的“UE4CtrlAPI”类可以将发送端的端口与IP地址进行实例化：

import UE4CtrlAPI as UE4CtrlAPI

ue = UE4CtrlAPI.UE4CtrlAPI()

关键功能的实现

常用python接口详解

sendUE4Cmd

接口函数解释

def sendUE4Cmd(self, cmd, windowID=-1):

"""send command to control the display style of RflySim3D

struct Ue4CMD0{

int checksum;

char data[52];

} i52s

struct Ue4CMD{

int checksum;

char data[252];

} i252s

"""

这个python接口定义了一个名为sendUE4Cmd的函数，它接受两个参数：cmd和windowID。cmd参数是一个字符串或字节对象，它包含一个用于控制RflySim3D的显示样式的命令，RflySim3D是一个用于无人机的三维仿真软件。windowID参数是一个整数，它指定了要发送命令的窗口。如果windowID是-1，那么命令会发送到所有的窗口。

if isinstance(cmd, str):

cmd = cmd.encode()

函数首先检查cmd参数的类型。如果它是一个字符串，就把它转换成字节。

if len(cmd) <= 51:

buf = struct.pack("i52s", 1234567890, cmd)

elif len(cmd) <= 249:

buf = struct.pack("i252s", 1234567890, cmd)

else:

print("Error: Cmd is too long")

return

然后检查cmd参数的长度，如果它的长度超过了249字节，就打印一个错误信息并返回。否则，就用struct模块把cmd参数打包成一个二进制缓冲区，同时加上一个校验值1234567890。缓冲区的格式取决于cmd参数的长度。如果它小于或等于51字节，缓冲区的格式是"i52s"，表示一个int后面跟着一个大小为52的char数组。如果它在52到249字节之间，缓冲区的格式是"i252s"，表示一个int后面跟着一个大小为252的char数组。

if windowID < 0:

for i in range(3): # 假设最多开了三个窗口

self.udp_socket.sendto(buf, (self.ip, 20010 + i))

else:

self.udp_socket.sendto(

buf, (self.ip, 20010 + windowID)

)

函数然后用socket模块把缓冲区发送到UDP套接字。套接字的地址取决于windowID和self对象的ip属性，self对象是RflySim类的一个实例。如果windowID是-1，缓冲区会发送到同一个ip地址上从20010开始的三个连续的端口。如果windowID不是-1，缓冲区会发送到ip地址上的20010
+
windowID的端口。ip地址可以是一个单播地址（例如，"127.0.0.1"或"192.168.1.100"）或一个多播地址（例如，"224.0.0.10"或"255.255.255.255"）。

if "RflyChangeMap" in cmd.decode():

time.sleep(0.5)

函数还检查cmd参数是否包含了子字符串"RflyChangeMap"，这表示一个改变仿真中场景的命令。如果是这样，它在发送缓冲区后等待0.5秒，以便场景完全切换。

发送示例

ue.sendUE4Cmd('RflyChangeMapbyName Grasslands')

利用控制台命令RflyChangeMapbyName将地图切换到Grasslands

fillList

接口函数解释

def fillList(self,data,inLen):

这个函数的作用是将输入的数据转换成一个指定长度的列表。它接受两个参数：data和inLen。data可以是一个numpy数组、一个列表或者一个单一的数值，inLen是一个整数，表示期望的列表长度。

if isinstance(data, np.ndarray):

data = data.tolist()

首先，函数检查data是否是一个numpy数组，如果是的话，就将其转换成一个普通的列表。

if isinstance(data, list) and len(data)==inLen:

return data

else:

if isinstance(data, list):

datLen = len(data)

if datLen<inLen:

data = data + [0]* (inLen-datLen)

if datLen>inLen:

data = data[0:inLen]

else:

data = [data] + [0]* (inLen-1)

return data

然后，函数检查data是否是一个列表，并且长度是否和inLen相等，如果是的话，就直接返回data。否则，函数会根据data的类型和长度进行不同的处理：

- 如果data是一个列表，但长度小于inLen，就在列表后面补充0，直到长度等于inLen。

- 如果data是一个列表，但长度大于inLen，就截取列表的前inLen个元素。

-
如果data不是一个列表，就将data作为列表的第一个元素，然后在后面补充0，直到长度等于inLen。

最后，函数返回转换后的列表。

sendUE4Attatch

接口函数解释

def sendUE4Attatch(self, CopterIDs, AttatchIDs, AttatchTypes, windowID=-1):

"""Send msg to UE4 to attach a vehicle to another (25 vehicles);

CopterIDs,AttatchIDs,AttatchTypes can be a list with max len 25

"""

struct VehicleAttatch25 {

int checksum;//1234567892

int CopterIDs[25];

int AttatchIDs[25];

# int
AttatchTypes[25];//0：正常模式，1：相对位置不相对姿态，2：相对位置+偏航（不相对俯仰和滚转），3：相对位置+全姿态（俯仰滚转偏航）

}i25i25i25i

sendUE4Attatch函数的作用是向UE4发送消息，让一个飞行器附着在另一个飞行器上。该函数有四个参数：self,
CopterIDs, AttatchIDs, AttatchTypes, 和
windowID。self参数表示调用该函数的类的实例。CopterIDs参数是一个整数列表，表示要附着的飞行器的ID。AttatchIDs参数是一个整数列表，表示要附着在的飞行器的ID。AttatchTypes参数是一个整数列表，表示每对飞行器的附着类型。windowID参数是一个可选的整数，表示要发送消息的窗口的ID。

change the 1D variable to 1D list

CopterIDs=self.fillList(CopterIDs,25)

AttatchIDs=self.fillList(AttatchIDs,25)

AttatchTypes=self.fillList(AttatchTypes,25)

该函数首先调用fillList方法，确保三个列表CopterIDs, AttatchIDs, 和
AttatchTypes的长度都是25。如果任何一个列表的长度小于25，它会在末尾补零。如果任何一个列表的长度大于25，它会截取前25个元素。这是因为该函数使用的消息格式只能容纳25对飞行器。

buf = struct.pack(

"i25i25i25i", 1234567892, *CopterIDs, *AttatchIDs, *AttatchTypes

)

然后使用struct模块将四个参数打包成一个二进制缓冲区。缓冲区的格式如下：

- 一个整数校验和，值为1234567892

- 一个25个元素的整数数组，表示CopterIDs

- 一个25个元素的整数数组，表示AttatchIDs

- 一个25个元素的整数数组，表示AttatchTypes

if windowID < 0:

for i in range(3): # 假设最多开了三个窗口

self.udp_socket.sendto(buf, (self.ip, 20010 + i))

else:

self.udp_socket.sendto(

buf, (self.ip, 20010 + windowID)

) # specify PC's IP to send

该函数然后将缓冲区发送到UDP套接字，IP地址和端口号由self对象的ip和windowID属性指定。如果windowID是负数，该函数会将缓冲区发送到所有三个窗口（假设有三个窗口打开），通过在端口号上加上20010,
20011, 和
20012的值。如果windowID不是负数，该函数会将缓冲区发送到只有一个窗口，端口号等于20010加上windowID。

发送示例

“PythonSendUE4Attatch.py”文件代码如下：

```
import time

import UE4CtrlAPI as UE4CtrlAPI

ue = UE4CtrlAPI.UE4CtrlAPI()

ue.sendUE4PosNew(1,3,[0,0,-10],[0,0,0],[0,0,0],[0,0,0,0,0,0,0,0])

ue.sendUE4PosNew(2,3,[0,0,0],[0,0,0],[0,0,0],[0,0,0,0,0,0,0,0])

ue.sendUE4Attatch(2,1,3)

ue.sendUE4PosNew(2,3,[1,0,0],[0,0,0],[0,0,0],[0,0,0,0,0,0,0,0])

ue.sendUE4Cmd(b"RflyChangeViewKeyCmd S -1")

ue.sendUE4Cmd(b"RflyChangeViewKeyCmd N -1")

theta=0

posX=0

while True:

ue.sendUE4PosNew(1,3,[posX,0,-10],[0,0,theta],[0,0,0],[0,0,0,0,0,0,0,0])

theta=(theta+0.1)%360

posX=posX+0.1

time.sleep(0.1)
```

这串代码先用两个sendUE4PosNew函数创建了两个四旋翼飞机，ID分别为1与2。然后是使用sendUE4Attatch函数以3号附加模式将2号飞机附加到1号飞机上。然后再使用了一个sendUE4PosNew函数重新设置2号飞机的位置（此时设置的是2号飞机相对于1号飞机的位置，设置在了1号飞机前方1米处）。

然后给RflySim3D发送了两个按键命令[1]，相当于手动在RflySim3D中按下S键、N键，-1表示不加数字键，（在3.2.3中介绍过S键是显示飞机的ID，N键是启动上帝视角，这样视角就不会跟着飞机转了，否则容易头晕）。

接下来是一个while
true的循环，可以看见该程序每隔0.1s就向RflySim3D发送一次1号飞机的坐标，每次1号飞机的x坐标会增大0.1米，绕z轴的旋转角yaw增大0.1度。

可以看见，我们并没有更新2号飞机的位置，但是2号飞机仍在运动，这是因为它已经被“附加在1号飞机上”了，1号飞机运动时2号飞机会自动跟随运动。

sendUE4ExtAct

接口函数解释

```
def sendUE4ExtAct(

self,

copterID=1,

ActExt=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

windowID=-1,

):
```

这个函数有四个参数：

- self: 一个对象，包含了套接字的属性，IP地址，和开始时间

- copterID: 要发送的飞行器的ID，缺省值为1

- ActExt: 外部动作的值的列表，缺省值为全零

- windowID: 要发送消息的窗口的ID，缺省值为-1

该函数用UDP发送的结构体形式如下：

struct Ue4ExtMsg {

int checksum;//1234567894

int CopterID;

double runnedTime; //Current stamp (s)

double ExtToUE4[16];

}

它是一个长度为4×1+4×1+8×1+8×16=144字节的结构体

这个函数向一个UDP套接字发送一条消息，包含了一个飞行器的外部动作信息。这条消息有以下几个字段：

- checkSum: 一个固定的值1234567894，用于验证消息的完整性

- copterID: 一个整数，表示飞行器的ID，缺省值为1

- runnedTime: 一个双精度浮点数，表示从程序开始运行到现在的时间，单位是秒

- ExtToUE4: 一个包含16个双精度浮点数的列表，表示外部动作的值，缺省值为全零

ActExt=self.fillList(ActExt,16)

runnedTime = time.time() - self.startTime

checkSum = 1234567894

buf = struct.pack("2i1d16d", checkSum, copterID, runnedTime, *ActExt)

这个函数首先检查ActExt列表的长度，如果小于16，就用零填充，然后用struct模块把字段打包成一个二进制缓冲区。

if windowID < 0:

for i in range(3): # 假设最多开了三个窗口

self.udp_socket.sendto(buf, (self.ip, 20010 + i))

else:

self.udp_socket.sendto(

buf, (self.ip, 20010 + windowID)

)

这个函数再把缓冲区发送到UDP套接字，套接字的IP地址和端口号由self对象的ip和windowID属性决定。如果windowID是负数，这个函数就把缓冲区发送到所有三个窗口（假设最多开了三个窗口），通过在端口号上加上20010，20011，和20012。如果windowID不是负数，这个函数就把缓冲区发送到一个指定的窗口，其端口号等于20010加上windowID。

发送示例

ue.sendUE4ExtAct(1000,[0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0])

调用ID为1000的无人机的蓝图接口“ActuatorInputsExt”，并传入后续16维数组作为它的参数。而该飞机传入参数的第8位如果是1，会触发它的爆炸逻辑。

其实根据sendUE4Cmd的内容，我们可以知道如果使用如下代码：

ue.sendUE4Cmd('RflySetActuatorPWMsExt 1000 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0')

完全可以达成同样的效果，在RflySim3D控制台命令接口中[2]，我们介绍过该命令接口，它的作用也是调用无人机的蓝图接口“ActuatorInputsExt”，和sendUE4ExtAct函数作用是一样的。那么如果通过上一节中的命令，将整个控制台命令发送给RflySim3D，那么它也是可以触发飞机的爆炸效果的。

sendUE4PosNew

接口函数解释

向RflySim3D中指定id的模型发送样式、位置、执行器和角度信息，以创建新的3D模型或更新旧模型的状态

def sendUE4PosNew(

self,

copterID=1,

vehicleType=3,

PosE=[0, 0, 0],

AngEuler=[0, 0, 0],

VelE=[0, 0, 0],

PWMs=[0] * 8,

runnedTime=-1,

windowID=-1,

):

sendUE4PosNew 函数的输入参数包括：

1. **copterID**: 默认值为1。表示无人机的ID号，用于区分不同的无人机。
2. **vehicleType**: 默认值为3。表示载具的类型，其中3表示四旋翼无人机。
3. **PosE**: 默认值为[0, 0, 0]。表示无人机在地球坐标系中的位置，包括x、y、z三个方向。
4. **AngEuler**: 默认值为[0, 0, 0]。表示无人机的欧拉角，包括滚动、俯仰、偏航三个角度。
5. **VelE**: 默认值为[0, 0, 0]。表示无人机在地球坐标系中的速度，包括x、y、z三个方向。这里的速度没有作用，因为RflySim3D并不负责进行运动学解算
6. **PWMs**: 默认值为[0] * 8。表示电机的PWM信号，总共有8个电机。
7. **runnedTime**: 默认值为-1。表示无人机的运行时间，单位为秒。如果为-1，则表示获取当前时间作为运行时间。
8. **windowID**: 默认值为-1。表示要发送数据的模拟器窗口ID。如果为-1，则发送数据到所有打开的窗口。

struct SOut2SimulatorSimpleTime {

int checkSum; //1234567891

int copterID; //Vehicle ID

int vehicleType; //Vehicle type

int PosGpsInt[3]; //lat*10^7,lon*10^7,alt*10^3，int型发放节省空间

float MotorRPMS[8];

float VelE[3];

float AngEuler[3]; //Vehicle Euler angle roll pitch yaw (rad) in x y z

double PosE[3]; //NED vehicle position in earth frame (m)

double runnedTime; //Current Time stamp (s)

}6i14f4d

输出到模拟器的数据结构包含以下字段：

- checkSum: 一个整数，用于验证数据的有效性。它应该等于1234567891。

- copterID: 一个整数，用于在模拟器中区分不同的飞行器。

- vehicleType: 一个整数，用于指示飞行器的类型，如四旋翼、固定翼或直升机。

- PosGpsInt:
一个长度为三的整数数组，表示飞行器在全球坐标系中的纬度、经度和高度。单位是纬度和经度乘以10^7，高度乘以10^3。这种格式可以减小数据的大小。

- MotorRPMS:
一个长度为八的浮点数数组，表示飞行器的各个电机的转速，单位是每分钟转数(RPM)。

- VelE:
一个长度为三的浮点数数组，表示飞行器在地球坐标系中的速度，该坐标系与北东下(NED)轴对齐。单位是米每秒。

- AngEuler:
一个长度为三的浮点数数组，表示飞行器在地球坐标系中的欧拉角，该角度描述了飞行器相对于NED轴的方向。角度分别是横滚、俯仰和偏航，单位是弧度。

- PosE:
一个长度为三的双精度浮点数数组，表示飞行器在地球坐标系中的位置，该坐标系与NED轴对齐。单位是米。

- runnedTime: 一个双精度浮点数，表示模拟器的当前时间戳，单位是秒。

该结构体的总大小是6个整数，14个浮点数和4个双精度浮点数，相当于104个字节。该结构体可以使用字符串"6i14f4d"作为格式说明符，打包成二进制格式。

PosE=self.fillList(PosE,3)

AngEuler=self.fillList(AngEuler,3)

VelE=self.fillList(VelE,3)

PWMs=self.fillList(PWMs,8)

首先填充列表

PosGpsInt=[0,0,0]

checkSum = 1234567891

buf = struct.pack(

"6i14f4d",

checkSum,

copterID,

vehicleType,

*PosGpsInt,

*PWMs,

*VelE,

*AngEuler,

*PosE,

runnedTime

)

打包为SOut2SimulatorSimpleTime格式

if windowID < 0:

for i in range(3): # 假设最多打开三个窗口

self.udp_socket.sendto(buf, (self.ip, 20010 + i))

else:

self.udp_socket.sendto(

buf, (self.ip, 20010 + windowID)

)

根据windowID选择发送的目标端口

发送示例

ue.sendUE4PosNew(10,3,[0,0,-10],[0,0,0],[0,0,0],[0,0,0,0,0,0,0,0])

发送了一个ID为10的飞机的数据，它的ClassID为3，表示我们创建的是一个四旋翼无人机。对象的位置是[0,0,-10]，意味着它位于坐标系原点上方10米处。对象的姿态和速度都是[0,0,0],；所有螺旋桨转速为0，意味着它没有相对于全局轴旋转且对象是静止的。

快速布置场景

Bat脚本添加模型

在LoadModelsOnBat.bat脚本中通过如下命令调用UEImportScript.py

start %PSP_PATH%\Python38\python.exe "%~dp0\UEImportScript.py"

这个命令启动一个新的进程，运行来自环境变量PSP_PATH指定的路径的Python 3.8。执行的Python脚本是UEImportScript.py，它位于与批处理脚本相同的目录（%~dp0）。

通过UEImportScript.py调用sendUE4Pos向RflySim3D发送添加模型到场景中的命令

import time

import math

import sys

import UE4CtrlAPI as UE4CtrlAPI

首先导入必要的依赖库文件

ue = UE4CtrlAPI.UE4CtrlAPI()

调用UE4CtrlAPI.py库文件下的UE4CtrlAPI类创建一个通信实例ue。

ue.sendUE4Pos(100,30,0,[2.5,0,-8.086],[0,0,math.pi])

向RflySim3D发送udp消息，控制初始位置和姿态生成3D对象。sendUE4Pos函数在UE4CtrlAPI.py库文件中的完整定义

sendUE4Pos(self,copterID=1,vehicleType=3,MotorRPMSMean=0,PosE=[0,0,0],AngEuler=[0,0,0],windowID=-1)

其中车辆ID为CopterID=100;模型类别VehicleType=30(人物);电机转速MotorRPMSMean=0;位置坐标PosM=[2.5,0，-8.086]，单位m;飞行器姿态角AngEulerRad=[0,0,math.pi]，单位rad(math.pi即绕z轴旋转180度面向屏幕前显示);UDP消息接收窗口号默认为windowsID=-1(发送到所有打开的RflySim3D程序)。模型类别（vehicleType）选择:四轴飞行器3，六轴飞行器5/6，人30，棋盘格平台40，汽车50/51，灯60，固定翼飞机100，圆环方框类靶标150/152。

ue.sendUE4PosScale(101,200030,0,[10.5,0,-8.086],[0,0,math.pi],[10,10,10])

该函数作用与sendUE4Pos相似，也是发送三维模型的数据，只是更新的数据有所不同，它额外发送了一个缩放数据Scale，可以控制三维模型的显示样式大小。

ue.sendUE4Cmd('RflyChange3DModel 100 12')

调用RflySim3D控制台命令' RflyChange3DModel 100 12'修改三维模型显示样式。这里的RflyChange3DModel命令表示将copterID为100的模型样式修改为vehicleType为12的行人。

加载txt文件布置场景

在Grasslands.txt中预存控制台命令

CreateVehicle, 1000, 3, 0.66, -0.11, -8.15, 0.00, 0.00, -0.09

这条命令在仿真环境中创建了一个ID为1000，模型为3（四旋翼飞行器），初始位置为（0.66, -0.11, -8.15），初始姿态为（0.00, 0.00, -0.09）的飞行器。

CreateVehicle, 1001, 30, 1.88, -0.60, -8.23, 0.00, 0.00, -0.24

这条命令在仿真环境中创建了一个ID为1001，模型为30（飞机），初始位置为（1.88, -0.60, -8.23），初始姿态为（0.00, 0.00, -0.24）的飞行器。

CreateVehicle, 1002, 50, 3.56, 1.65, -8.12, 0.00, 0.00, 0.10

这条命令在仿真环境中创建了一个ID为1002，模型为50（直升机），初始位置为（3.56, 1.65, -8.12），初始姿态为（0.00, 0.00, 0.10）的飞行器。

RflyMoveVehiclePosAng 1000 1 0.00 2.00 0.00 0.00 0.00

这条命令将ID为1000的飞行器相对于其当前位置和姿态，以模式1（线性插值）移动到新的位置（0.00, 2.00, 0.00）和姿态（0.00, 0.00, 0.00）。

通过LoadModelsByTxt.py读取txt中的控制台命令并调用sendUE4Cmd发送给RflySim3D来创建模型

import time

import math

import sys

import UE4CtrlAPI as UE4CtrlAPI

首先导入必要的依赖库文件

ue = UE4CtrlAPI.UE4CtrlAPI()

调用UE4CtrlAPI.py库文件下的UE4CtrlAPI类创建一个通信实例ue。

ue.sendUE4Cmd(b'RflyChangeMapbyName Grasslands')

调用RflySim3D控制台命令'RflyChangeMapbyName Grasslands'修改UE场景。这里的RflyChangeMapbyName命令表示切换地图(场景)，后面的字符串是地图名称，这里会将所有打开的窗口切换为草地地图。sendUE4Cmd函数在UE4CtrlAPI.py库文件中的完整定义

sendUE4Cmd(cmd, windowid =-1)

其中cmd为命令字符串，windowid为接收窗口号(假设同时打开多个RflySim3D窗口)，windowid =-1表示默认发送到所有窗口。

imPath = os.path.join(sys.path[0],'Grasslands.txt')

str = 'RflyLoad3DFile ' + imPath

print(str)

ue.sendUE4Cmd(str.encode())

print(str.encode())

此处使用RflyLoad3DFile命令来加载txt格式场景创建文件。

imPath = os.path.join(sys.path[0],'Grasslands.txt')：

这行代码使用os.path.join函数将当前脚本所在的目录（通过sys.path[0]获取）与文件名 'Grasslands.txt' 组合成一个完整的文件路径，并将结果存储在imPath变量中。

1. str = 'RflyLoad3DFile ' + imPath：

这行代码将构建一个字符串，其中包含 RflyLoad3DFile 命令和 imPath
变量中的文件路径，这是要发送给UE的命令。

i. print(str)：
这行代码用于在控制台输出构建的命令字符串，以便进行调试和查看。

ii. ue.sendUE4Cmd(str.encode())：
这行代码使用 sendUE4Cmd
方法将构建的命令字符串编码为字节并发送给UE模拟环境。

iii. print(str.encode())：
这行代码用于在控制台输出编码后的命令字符串。在这里，打印了命令的字节表示，以便查看发送给UE的确切数据。

Python布置动态场景

1. 导入必要的依赖库文件

import time

import math

import sys

import UE4CtrlAPI as UE4CtrlAPI

import UEMapServe

首先导入必要的依赖库文件

2. 调用RflySim3D场景控制接口类

ue = UE4CtrlAPI.UE4CtrlAPI()

调用UE4CtrlAPI.py库文件下的UE4CtrlAPI类创建一个通信实例ue。

ue.sendUE4Cmd(b'RflyChangeMapbyName Grasslands')

调用RflySim3D控制台命令'RflyChangeMapbyName
Grasslands'修改UE场景。这里的RflyChangeMapbyName命令表示切换地图(场景)，后面的字符串是地图名称，这里会将所有打开的窗口切换为草地地图。sendUE4Cmd函数在UE4CtrlAPI.py库文件中的完整定义

PosInit=[0,0,-8.086]

ue.sendUE4Pos(1,3,0,PosInit,[0,0,0])

向RflySim3D发送udp消息，控制初始位置（使用北东地坐标系）生成3D对象，copterID为1的四旋翼。

3. 调用RflySim3D地形服务接口类

map = UEMapServe.UEMapServe('Grasslands')

创建一个地形服务器的类，并加载地图Grasslands的地形数据（本目录的png和txt文件）

x=1

y=1

z = map.getTerrainAltData(x,y)

获取本地地形高度

ue.sendUE4Pos(2,3,0,[x,y,z],[0,0,0])

根据上一步指定位置的地形高度，在该位置创建贴合地面的物体，copterID为2的四旋翼

ue.sendUE4PosScale2Ground(100,200030,0,[3,0,-100],[0,0,math.pi],[1,1,1])

此方法会自动调用getTerrainAltData计算指定位置的地形高度z，生成贴合地面的物体，故这里给出的高度-100（NED坐标系）可以为任意值。注意：在地形层数比较复杂的地方，位置z的默认值应该在地形稍上方，避免贴合在错误表皮上

4. 构建一个死循环不断更新无人机的位置和姿态

while True:

t=time.time()-startTime

动态设置多旋翼的位置

x= math.sin(t*0.1)+PosInit[0]

y= math.cos(t*0.1)+PosInit[1]

z= -t*0.1+PosInit[2]

设定多旋翼姿态

p=math.sin(t*0.01)/10

q=math.sin(t*0.01)/10

r=math.sin(t*0.01)/10

发送下一时刻多旋翼位置和姿态到场景

ue.sendUE4Pos(1,3,1000,[x,y,z],[p,q,r])

相机视角调整

UE4ViewPortDemo.py调用sendUE4Cmd发送控制台命令调整相机视角

ue = UE4CtrlAPI.UE4CtrlAPI()

调用UE4CtrlAPI.py库文件下的UE4CtrlAPI类创建一个通信实例ue。

ue.sendUE4Cmd('RflyChangeMapbyName Grasslands')

调用RflySim3D控制台命令'RflyChangeMapbyName Grasslands'修改UE场景。这里的RflyChangeMapbyName命令表示切换地图(场景)，后面的字符串是地图名称，这里会将所有打开的窗口切换为草地地图。sendUE4Cmd函数在UE4CtrlAPI.py库文件中的完整定义

sendUE4Cmd(cmd, windowid =-1)

其中cmd为命令字符串，windowid为接收窗口号(假设同时打开多个RflySim3D窗口)，windowid =-1表示发送到所有窗口。

PosInit=[0,0,-8.086]

ue.sendUE4Pos(1,3,0,PosInit,[0,0,0])

创建物体并初始化物体的位置

ue.sendUE4PosScale2Ground(100,200030,0,[3,0,-100],[0,0,math.pi],[1,1,1])

创建自动贴合地面的物体，可设置其显示尺寸。

完成上述步骤后，在python程序中调用sendUE4Cmd接口，参数设置需要执行的[1]RflySim3D控制台指令（相机控制相关接口包括RflyChangeViewKeyCmd、RflyCameraPosAng、RflyCameraPosAngAdd、RflyCameraFovDegrees）。

浮动可视化信息接口

UE4MsgDispDemo.py调用sendUE4Cmd发送控制台命令设置模型的标签

首先导入必要的依赖库文件

ue = UE4CtrlAPI.UE4CtrlAPI()

调用UE4CtrlAPI.py库文件下的UE4CtrlAPI类创建一个通信实例ue。

ue.sendUE4Cmd(b'RflyChangeMapbyName Grasslands')

调用RflySim3D控制台命令'RflyChangeMapbyName Grasslands'修改UE场景。这里的RflyChangeMapbyName命令表示切换地图(场景)，后面的字符串是地图名称，这里会将所有打开的窗口切换为草地地图。sendUE4Cmd函数在UE4CtrlAPI.py库文件中的完整定义

sendUE4Cmd(cmd, windowid =-1)

其中cmd为命令字符串，windowid为接收窗口号(假设同时打开多个RflySim3D窗口)，windowid =-1表示发送到所有窗口。

PosInit=[0,0,-8.086]

ue.sendUE4Pos(1,3,0,PosInit,[0,0,0])

创建物体并初始化物体的位置

ue.sendUE4PosScale2Ground(100,200030,0,[3,0,-100],[0,0,math.pi],[1,1,1])

创建自动贴合地面的物体，可设置其显示尺寸。

完成上述步骤后，在python程序中调用sendUE4Cmd接口，参数设置需要执行的[1]RflySim3D控制台指令（模型标签设置相关接口包括RflySetMsgLabel、RflySetIDLabel）），运行python脚本发送命令后即可设置目标copter的标签属性。

触发载具爆炸特效

UE4CtrlAPITest.py调用sendUE4PosNew创建蓝图模型并调用sendUE4ExtAct触发蓝图中定义的爆炸特效

import time

import math

import sys

import UE4CtrlAPI as UE4CtrlAPI

首先导入必要的依赖库文件

ue = UE4CtrlAPI.UE4CtrlAPI()

调用UE4CtrlAPI.py库文件下的UE4CtrlAPI类创建一个通信实例ue。

ue.sendUE4Cmd(b'RflyChangeMapbyName Grasslands')

调用RflySim3D控制台命令'RflyChangeMapbyName Grasslands'修改UE场景。这里的RflyChangeMapbyName命令表示切换地图(场景)，后面的字符串是地图名称，这里会将所有打开的窗口切换为草地地图。sendUE4Cmd函数在UE4CtrlAPI.py库文件中的完整定义

sendUE4Cmd(cmd, windowid =-1)

其中cmd为命令字符串，windowid为接收窗口号(假设同时打开多个RflySim3D窗口)，windowid =-1表示发送到所有窗口。

ue.sendUE4PosNew(100,208,[0,0,-8.086],[0,0,0],[0,0,0],[0,0,0,0,0,0,0,0,0])

创建一个VehicleType（classID）为208的EastBomberSU24战斗机。

ue.sendUE4ExtAct(100,[0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0])

触发EastBomberSU24战斗机的爆炸特效。

UDP通信验证

initUE4MsgRec（初始化UDP监听）

这个接口定义了一个函数，用于初始化从UE4接收UDP数据

def initUE4MsgRec(self):

""" Initialize the UDP data linsening from UE4,

currently, the crash data is listened

"""

self.stopFlagUE4=False

#print("InitUE4MsgRec", self.stopFlagUE4)

self.inSilVect = []

self.inReqVect = []

self.inReqUpdateVect = []

MYPORT = 20006

MYGROUP = '224.0.0.10'

ANY = '0.0.0.0'

self.udp_socketUE4.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)

self.udp_socketUE4.bind((ANY,MYPORT))

status = self.udp_socketUE4.setsockopt(socket.IPPROTO_IP,

socket.IP_ADD_MEMBERSHIP,

socket.inet_aton(MYGROUP) + socket.inet_aton(ANY))

self.t4 = threading.Thread(target=self.UE4MsgRecLoop, args=())

self.t4.start()

- 首先，将`self.stopFlagUE4`设置为False，表示不停止UE4消息的接收。
- 然后，创建三个空列表`self.inSilVect`、`self.inReqVect`和`self.inReqUpdateVect`，用于存储接收到的数据。
- 接下来，定义了常量`MYPORT`和`MYGROUP`，分别表示端口号和组播地址。
- 然后，将`ANY`设置为`'0.0.0.0'`，表示接收来自任何IP地址的数据。
- 接着，通过`setsockopt`方法设置了一些Socket选项，其中`socket.SO_REUSEADDR`表示允许地址重用。
- 然后，通过`bind`方法将UDP套接字绑定到指定的IP地址和端口号。
- 接下来，通过`setsockopt`方法将套接字加入到指定的IP组中，以便接收组内的数据。
- 最后，创建了一个新的线程`t4`，并将其绑定到`UE4MsgRecLoop`方法上，并启动该线程。这个方法将负责接收UE4发来的消息。

endUE4MsgRec（结束UE4消息监听）

endUE4MsgRec用于停止UE4MsgRecLoop方法的执行，并关闭UDP套接字。

def endUE4MsgRec(self):

"""End UE4 message listening"""

self.stopFlagUE4 = True

time.sleep(0.5)

self.t4.join()

self.udp_socketUE4.close()

- 首先，它将属性self.stopFlagUE4设为True，这个属性是一个布尔值，用于控制UE4MsgRecLoop方法的死循环。当self.stopFlagUE4为True时，UE4MsgRecLoop方法会跳出循环，结束线程。
- 然后，它调用time.sleep(0.5)方法，暂停0.5秒，以确保UE4MsgRecLoop方法能够正常退出。
- 接着，它调用self.t4.join()方法，等待UE4MsgRecLoop方法的线程t4结束，释放资源。
- 最后，它调用self.udp_socketUE4.close()方法，关闭与UE4通信的UDP套接字，断开连接。

UE4MsgRecLoop（UDP侦听循环）

UE4MsgRecLoop定义了一个死循环，用于不断地监听来自UE4的UDP数据。

def UE4MsgRecLoop(self):

这个死循环的逻辑如下：

while True:

if self.stopFlagUE4:

break

首先检查self.stopFlagUE4是否为真，如果是，则跳出循环，停止监听。

try:

buf, addr = self.udp_socketUE4.recvfrom(65500)

然后尝试从self.udp_socketUE4接收UDP数据包，并存储在buf和addr中。buf是接收到的数据，addr是发送端的地址。

监听CopterSimCrash

struct CopterSimCrash {

int checksum;

int CopterID;

int TargetID;

}

CopterSimCrash结构体包含以下字段：

- checksum: 一个校验码，固定为1234567890，用于验证数据的有效性

- CopterID: 一个整数，表示发生碰撞的无人机的编号

- targetID: 一个整数，表示被碰撞的目标的编号，如果没有目标，则为-1

if len(buf) == 12:

checksum, CopterID, targetID = struct.unpack("iii", buf[0:12])

if checksum == 1234567890:

if targetID > -0.5 and CopterID == self.CopterID:

self.isVehicleCrash = True

self.isVehicleCrashID = targetID

print(

"Vehicle #", CopterID, " Crashed with vehicle #", targetID

)

接着检查buf的长度。如果是12字节，就意味着数据包包含三个整数：checksum, CopterID,
和 targetID。使用struct模块解包这些值。