RflySim3D 常用场景控制接口实验原理

1. 文件目录

例程目录：[安装目录]\RflySimAPIs\4.RflySimModel\0.ApiExps\14.inCopterData\

文件夹/文件名称	说明
1.UECtrlPy\Readme.pdf	常用 python 接口详解
2.LoadModelsOnBat\Readme.pdf	快速布置三维场景
3.LoadModelsByTxt\Readme.pdf	
4.TrajDemo\Readme.pdf	
7.DamageModel\Readme.pdf	触发载具爆炸特效
6.RflySim3DViewPortDemo\Readme.pdf	相机视角调整
8.RflySim3DMsgDispDemo\Readme.pdf	浮动可视化信息接口
9.RflySim3DPosGet\Readme.pdf	实时获取 RflySim3D 内所有物体位置、碰撞数据

2. 总体说明

2.1. 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。

2.1.1. 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。

2.1.2. 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()
```

3. 关键功能的实现

3.1. 常用 python 接口详解

3.1.1. 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

3.1.2. 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。

3.1.3. 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],[0,0,0,0,0,0,0,0,0])
ue.sendUE4PosNew(2,3,[0,0,0],[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],[0,0,0,0,0,0,0,0,0])
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],[0,0,0,0,0,0,0,0,0])
    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**错误!未找到引用源。**，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。

3.1.4. 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**错误!未找到引用源。**，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。

3.1.5. 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。

3.2. 快速布置场景

3.2.1. 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],wi
ndowID=-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。

3.2.2. 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。

```
a)   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。

```
b)   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。

```
c)   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。

```
d)   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。

3.2.3. 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])
```

3.3. 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）。

3.4. 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。

3.5. 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])
```

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。

3.6. UDP 通信验证

3.6.1. 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。

3.6.2. 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。

3.6.3. 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。

然后验证 checksum 是否等于 1234567890，这是一个预定义的值，用于验证数据包。如果 checksum 正确，检查 targetID 是否为正数，以及 CopterID 是否与当前飞行器的 ID 匹配。

如果两个条件都为真，设置标志 self.isVehicleCrash 和 self.isVehicleCrashID，表示当前飞行器与另一个飞行器发生了碰撞。还打印出两个涉及碰撞的飞行器的 ID。

监听 PX4SILIntFloat

```
                    if len(buf) == 120:
                        iValue = struct.unpack("10i20f", buf[0:120])
                        if iValue[0] == 1234567897:
                            isCopterExist = False
                            for i in range(len(self.inSilVect)):  # 遍历数据列表，飞机 ID 有没有出
现过
                                if self.inSilVect[i].CopterID == iValue[1]:  # 如果出现过，就直
接更新数据
                                    isCopterExist = True
                                    self.inSilVect[i].checksum = iValue[0]
                                    self.inSilVect[i].inSILInts = iValue[2:10]
                                    self.inSilVect[i].inSILFLoats = iValue[10:30]
                                    # break
                            if not isCopterExist:  # 如果没有出现过，就创建一个结构体
                                vsr = PX4SILIntFloat(iValue)
                                self.inSilVect = self.inSilVect + [
                                    copy.deepcopy(vsr)
                                ]  # 扩充列表，增加一个元素
```

检查缓冲区的长度是否为 120 字节，这意味着它包含 10 个整数和 20 个浮点数。然后它使用 struct 模块将缓冲区解包成一个包含 30 个值的元组。第一个值是一个校验和，应该等于 1234567897。第二个值是一个飞行器 ID，用于识别一个飞行器。其余的是与飞行器相关的传感器和控制数据。

维护一个 PX4SILIntFloat 对象的列表，每个对象表示一个具有自己 ID 和数据的飞行器。遍历列表，检查缓冲区中的飞行器 ID 是否与列表中的任何一个现有的飞行器匹配。如果匹配，就用缓冲区中的新数据更新相应的飞行器对象。如果不匹配，就用缓冲区中的数据创建一个新的飞行器对象，并将其追加到列表中。这样，就可以跟踪多个飞行器及其数据。

监听 reqVeCrashData

```
                    if len(buf) == 160:
                        isCopterExist = False
                        iValue = struct.unpack("4i1d29f20s", buf[0:160])
                        if iValue[0] == 1234567897:
                            # print(vsr.copterID,vsr.vehicleType)
                            for i in range(len(self.inReqVect)):  # 遍历数据列表，飞机 ID 有没有出
现过
                                if self.inReqVect[i].copterID == iValue[1]:  # 如果出现过，就直
接更新数据
                                    isCopterExist = True
                                    self.inReqVect[i].CopyData(
                                        iValue
                                    )  # =copy.deepcopy(vsr)
                                    self.inReqUpdateVect[i] = True
                                    break
                                    # break
                            if not isCopterExist:  # 如果没有出现过，就创建一个结构体
                                vsr = reqVeCrashData(iValue)
                                self.inReqVect = self.inReqVect + [
                                    copy.deepcopy(vsr)
                                ]  # 扩充列表，增加一个元素
                                self.inReqUpdateVect = self.inReqUpdateVect + [True]
```

然后尝试从 UDP 套接字接收 160 字节的数据，并将其存储在一个名为 buf 的变量中。如果 buf 的长度恰好是 160，那么表示数据是有效的，并且包含了一个车辆的信息。然后使用 struct 模块解包数据，并将其赋值给一个名为 iValue 的变量，它是一个包含 35 个元素的元组。iValue 的第一个元素是一个魔术数（1234567897），它表示数据包的开始。第二个元素是车辆的 ID，它是一个整数。其余的元素是车辆的数据字段，例如类型，时间，位置，速度，姿态等。

然后检查 self.inReqVect 列表中是否已经存在 ID 为 iValue[1]的车辆。如果存在，那么表示该车辆之前已经被检测到，且其信息需要被更新。设置一个名为 isCopterExist 的标志为 True，并使用 reqVeCrashData 类的 CopyData 方法将 iValue 中的数据复制到 self.inReqVect 列表中对应的实例中。还将 self.inReqUpdateVect 列表中对应的元素设置为 True，表示该车辆已经被更新。然后跳出 for 循环。

如果 self.inReqVect 列表中不存在 ID 为 iValue[1]的车辆，那么表示该车辆是新检测到的，且其信息需要被添加。设置 isCopterExist 标志为 False，并使用 iValue 中的数据创建一个 reqVeCrashData 类的新实例。然后将新实例追加到 self.inReqVect 列表中，并将一个 True 值追加到 self.inReqUpdateVect 列表中，表示该车辆已经被添加。然后继续 for 循环。

监听 CameraData

```
                    if len(buf) == 72:  # CameraData: #长度 72, 5i11f1d
                        # print('hello')
                        iValue = struct.unpack("5i11f1d", buf[0:72])
                        if iValue[0] == 1234567891:
                            isDataExist = False
```

```
                    # 按 SeqID 来构建相机列表
                    for i in range(len(self.CamDataVect)):  # 遍历数据列表，相机 ID 有没有出现过
                        if self.CamDataVect[i].SeqID == iValue[1]:  # 如果出现过，就直接更新数据
                            self.CamDataVect[i].CopyData(iValue)
                            isDataExist = True
                            break
                    if not isDataExist:  # 如果没有出现过，就创建一个结构体
                        vsr = CameraData(iValue)
                        self.CamDataVect = self.CamDataVect + [
                            copy.deepcopy(vsr)
                        ]  # 扩充列表，增加一个元素
```

如果 buf 的长度是 72，那么表示数据是有效的，并且包含了一个相机的信息。使用 struct 模块解包数据，并将其赋值给一个名为 iValue 的变量，它是一个包含 18 个元素的元组。iValue 的第一个元素是一个魔术数（1234567891），它表示数据包的开始。第二个元素是相机的序列号，它是一个整数。其余的元素是相机的数据字段，例如时间，位置，姿态，角度等。

然后检查 self.CamDataVect 列表中是否已经存在序列号为 iValue[1]的相机。如果存在，那么表示该相机之前已经被检测到，且其信息需要被更新。设置一个名为 isDataExist 的标志为 True，并使用 CameraData 类的 CopyData 方法将 iValue 中的数据复制到 self.CamDataVect 列表中对应的实例中。然>>后跳出 for 循环。

如果 self.CamDataVect 列表中不存在序列号为 iValue[1]的相机，那么表示该相机是新检测到的，且其信息需要被添加。设置 isDataExist 标志为 False，并使用 iValue 中的数据创建一个 CameraData 类的新实例。然后将新实例追加到 self.CamDataVect 列表中，表示该相机已经被添加。然后继续 for 循环。

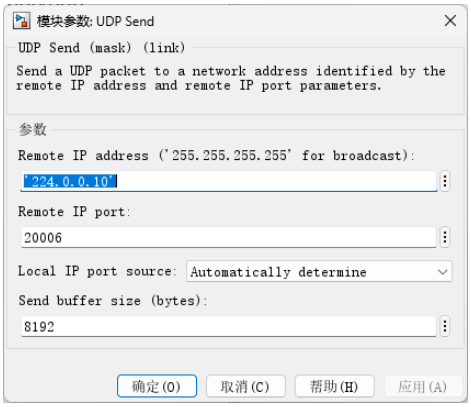
监听 CoptReqData

```
                if len(buf) == 80:  # CoptReqData: #长度 80, 2i16f1d
                    iValue = struct.unpack("2i16f1d", buf[0:80])
                    if iValue[0] == 1234567891:
                        isDataExist = False
                        # 按 CopterID 来构建物体飞机
                        for i in range(len(self.CoptDataVect)):  # 遍历数据列表，飞机 ID 有没有出现过
                            if (
                                self.CoptDataVect[i].CopterID == iValue[1]
                            ):  # 如果出现过，就直接更新数据
                                self.CoptDataVect[i].CopyData(iValue)
                                isDataExist = True
                                break
                        if not isDataExist:  # 如果没有出现过，就创建一个结构体
                            vsr = CoptReqData(iValue)
                            self.CoptDataVect = self.CoptDataVect + [
                                copy.deepcopy(vsr)
                            ]  # 扩充列表，增加一个元素
```

监听 ObjReqData

```
                    if len(buf) == 112:  # ObjReqData: #长度 112, 2i16f1d32s
                        iValue = struct.unpack("2i16f1d32s", buf[0:112])
                        if iValue[0] == 1234567891:
                            isDataExist = False
                            # 按 reqID 来构建物体列表
                            for i in range(len(self.ObjDataVect)):  # 遍历数据列表，物体 ID 有没有
出现过
                                if self.ObjDataVect[i].seqID == iValue[1]:  # 如果出现过，就直接
更新数据
                                    self.ObjDataVect[i].CopyData(iValue)
                                    isDataExist = True
                                    break
                            if not isDataExist:  # 如果没有出现过，就创建一个结构体
                                vsr = ObjReqData(iValue)
                                self.ObjDataVect = self.ObjDataVect + [
                                    copy.deepcopy(vsr)
                                ]  # 扩充列表，增加一个元素
```

3.6.4. Simulink 发送 UDP



在 Simulink 中利用 UDP Send 模块向 20006 端口发送数据。

3.6.5. Python 监听 UDP

```
ue = UE4CtrlAPI.UE4CtrlAPI()
```

在 python 程序中调用 UE4CtrlAPI.py 库文件下的 UE4CtrlAPI 类创建一个通信实例 ue。

```
ue.initUE4MsgRec()
```

发送数据请求，然后 python 中就可以接收到 Simulink 模型中发送的数据。该方法定义 UDP 通信端口为 20006，多播组 IP 地址为'224.0.0.10'

3.7. 实时获取 RflySim3D 内所有物体位置、碰撞数据

3.7.1. getUE4Pos

这个函数的作用是获取 RflySim3D 仿真环境中一个 Copter 的位置。

```
    def getUE4Pos(self, CopterID=1):
```

getUE4Pos 函数接受一个名为 CopterID 的参数，默认值为 1。这个函数执行以下步骤：

```
        if self.stopFlagUE4:  # 如果没有启用监听程序
            self.initUE4MsgRec()
            time.sleep(1)
```

- 它检查 self 对象的 stopFlagUE4 属性是否为 True，这意味着没有启用从 Simulink 接收数据的监听程序。如果是这样，它调用 initUE4MsgRec 方法来初始化 UDP 通信端口和多播组 IP 地址，并等待一秒。

```
            for i in range(len(self.inReqVect)):  # 遍历数据列表，飞机 ID 有没有出现过
                if self.inReqVect[i].copterID == CopterID:
                    posE = self.inReqVect[i].PosE
                    return [posE[0], posE[1], posE[2], 1]
            return [0, 0, 0, 0]
```

- 它遍历 self 对象的 inReqVect 属性，这是一个包含飞行器 ID 和地球坐标系中位置的数据对象的列表。它将每个数据对象的飞行器 ID 与 CopterID 参数进行比较，如果匹配，它返回一个包含四个元素的列表：位置的 x，y，z 坐标，以及一个表示位置有效的标志 1。如果在 inReqVect 列表中找不到 CopterID 参数，它返回一个包含四个零的列表，表示位置无效或不可用。

3.7.2. getUE4Data

getUE4Pos 只返回 Copter 的位置坐标和一个有效标志，而 getUE4Data 返回 Copter 的整个数据对象，包括位置、欧拉角、速度、加速度等信息。

```
        def getUE4Data(self, CopterID=1):
```

getUE4Data 的函数，它接受一个参数 CopterID，表示要查询的 Copter 的 ID，默认值为 1。

```
            if self.stopFlagUE4:  # 如果没有启用监听程序
                self.initUE4MsgRec()
                time.sleep(1)
```

- 函数的第一行检查一个名为 stopFlagUE4 的属性，它是一个布尔值，表示是否启用了一个监听程序，用于从外部源接收数据。如果这个属性为真，说明监听程序没有启用，那么函数就调用另一个函数 initUE4MsgRec 来初始化监听程序，并等待一秒钟。

```
            for i in range(len(self.inReqVect)):  # 遍历数据列表，飞机 ID 有没有出现过
                if self.inReqVect[i].copterID == CopterID:
                    return self.inReqVect[i]
            return 0
```

- 函数的第二行开始一个循环，遍历 self 对象的 inReqVect 属性，这是一个包含飞行器 ID 和地球坐标系中位置的数据对象的列表。它将每个数据对象的飞行器 ID 与 CopterID 参数进行比较，如果匹配，它返回这个数据对象，包含位置的 x，y，z 坐标，以及一个表示位置有效的标志 1。如果在 inReqVect 列表中找不到 CopterID 参数，循环结束后，函数返回 0，表示位置无效或不可用。

3.7.3. GetUE4PosAPI.py 调用接口实时获取场景内物体状态

```
    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'RflyReqVehicleData 1')
```

发送消息给 RflySim3D，让其将当前收到的飞机数据转发出来，回传到组播地址 224.0.0.10 的 20006 端口。注：只有飞机位置发生改变时，才会将位置数据传出，因此本语句要放在最前面，确保后续创建的物体（Python 一次性创建）都能被传出

```
ue.initUE4MsgRec()
```

Python 开始飞机数据的监听，数据存储在 inReqUpdateVect 列表（是否更新标志），和 inReqVect 列表（碰撞数据）中。注意：监听语句应该放到 sendUE4Pos 系列语句之前，不然无法捕获创建的障碍物。

```
ue.sendUE4Pos(100,30,0,[2.5,0,-8.086],[0,0,math.pi])
ue.sendUE4PosScale(101,200030,0,[10.5,0,-8.086],[0,0,math.pi],[10,10,10])
```

创建 CopterID 分别为 100、101 的障碍物

```
TargetCopterID = 1
PosEF = ue.getUE4Pos(TargetCopterID)
```

通过 getUE4Pos 接口来获取 1 号飞机位置数据，这架飞机是启动软件在环脚本时自动创建的。

PosEF[4] = PosE[3]+Flag(是否有数据)

```
TargetCopterID = 100
PosEF = ue.getUE4Pos(TargetCopterID)
TargetCopterID = 101
PosEF = ue.getUE4Pos(TargetCopterID)
```

通过 getUE4Pos 接口来获取 100 号和 101 号障碍物位置数据

```
TargetCopterID = 102
PosEF = ue.getUE4Pos(TargetCopterID)
```

通过 getUE4Pos 接口来获取不存在的物体位置数据

```
TargetCopterID = 1
vsr = ue.getUE4Data(TargetCopterID)
```

通过 getUE4Data 函数获取 1 号 copter 的数据

下面的程序直接通过 inReqUpdateVect 列表和 inReqVect，定时检查接收到的数据是否有更新，并打印出更新后的数据。

```
lastTime = time.time()
num=0
lastClock=time.time()
lastCount=0
while True:
    lastTime = lastTime + 1/30.0
    sleepTime = lastTime - time.time()
    if sleepTime > 0:
        time.sleep(sleepTime)
    else:
        lastTime = time.time()

    for i in range(len(ue.inReqVect)):
        if ue.inReqUpdateVect[i]: # 如果 i 号数据有更新

print(ue.inReqVect[i].copterID,ue.inReqVect[i].PosE,ue.inReqVect[i].CrashType)
            ue.inReqUpdateVect[i]=False
```

inReqVect[i]列表中的 CrashType;//碰撞物体类型，-2 表示地面，-1 表示场景静态物体，

0 表示无碰撞，1 以上表示被碰飞机的 ID 号。这是在 reqVeCrashData 结构体**错误!未找到引用源。**中定义的。

4. 相关文献

[1]. ..\..\API.pdf

[2]. ..\e3_InitAPI\Intro.pdf

[3]. ..\e4_UAVCtrl\Intro.pdf

[4]. IConsoleManager | Unreal Engine Documentation

附加资源

官方文档：RflySim 官方文档：https://rflysim.com/doc/zh/

社区交流：加入 RflySim 技术交流群：951534390